Fachhochschule Köln
Cologne University of Applied Sciences
Campus Gummersbach
Fakultät für Informatik und Ingenieurwissenschaften

Fachhochschule Dortmund
University of Applied Sciences and Arts
Fachbereich Informatik

Verbundstudiengang Wirtschaftsinformatik

Masterthesis

(Sechs-Monats-Arbeit)

zur Erlangung

des Mastergrades

„Master of science“

in der Fachrichtung Informatik

Vergleich von Quadtree, kd-tree und r-tree für statische und dynamische Geodaten

Erstprüfer:	Prof. Dr. Heide Faeskorn-Woyke
Zweitprüfer:	Prof. Dr. Birgit Bertelsmeier
vorgelegt am:	01.11.2011
von cand.	Christopher Jung
aus	Rankenweg 10
	44265 Dortmund
E-Mail-Adresse:	bktheg@web.de
Tel.:	0231/467609
Matr.-Nr.:	7071746

Inhaltsverzeichnis

Abbildungsverzeichnis

Tabellenverzeichnis

Abkürzungsverzeichnis

BBox ..... Bounding Box
GIS ..... Geographisches Informationssystem
GPS ..... Global Positioning System
MBR ..... Minimum Bounding Rectangles
OGC ..... Open Geospatial Consortium
TMC ..... Traffic Message Channel

Kapitel 1

Einführung

1.1 Einleitung

Die Bedeutung von Geodaten hat in den letzten Jahren stark zugenommen. In immer mehr Bereichen des täglichen Lebens gibt es Anwendungen oder Dienste, die auf der Basis geographischer Informationen ihren Dienst verrichten. Gleichzeitig werden bereits bestehende Applikationen immer komplexer - aggregieren immer mehr Daten mit dem Ziel noch bessere Ergebnisse zu liefern.

Als Beispiel seien hier Navigationssysteme genannt. Diese wurden ursprünglich für Kraftfahrzeuge entwickelt und verbreiteten sich seit Ende der 1990er zunehmend vor allem im Bereich der PKWs. Basierend auf dem GPS-Signal werden dabei in Kombination mit Straßenkarten Routen zwischen einem Startpunkt und einem Endpunkt berechnet. Bei diesen Straßenkarten, die als Vektordaten vorliegen, handelt es sich um eine Form von Geodaten.

Im Laufe der vergangenen Jahre entwickelten sich nun diese Navigationssysteme in zwei für Konsumenten sichtbare Richtungen weiter. Zum einen wurden diese immer kleiner und in zunehmenden Maße in Mobiltelefone integriert. Dadurch wurde es dem Verbrauchern möglich, auch außerhalb des PKWs Navigationsdienste zu nutzen. Hier ist vor allem die jüngste Smartphone-Generation um die Betriebssysteme iOS und Android hervorzuheben. Zum anderen wurden die Navigationssysteme selbst immer intelligenter. Schrittweise wurden weitere Daten in die Routenberechnung integriert. Besonders auffällig hierbei sind die Stau- und Baustelleninformationen, welche mittels TMC und TMCpro[1] zu den Geräten übertragen werden. Die Informationen werden dann dynamisch in die Berechnung

[1] Bei TMC und TMCpro werden die Stau- und Baustelleninformationen mittels UKW in digitaler Form übertragen.

mit einbezogen. Der Verbraucher profitiert somit von einer günstigeren Route und kann somit solche nicht statischen Langsamfahrstellen gezielt meiden.

Aber auch diese Form der Wegfindung wurde inzwischen - unter Nutzung zusätzlicher geographischer Daten - weiter verbessert. Denn um einen Stau oder eine Baustelle an ein Navigationssystem melden zu können, muss diese ihrerseits an einen zentralen Dienst gemeldet werden. Im Falle von Baustellen ist dies in der Regel kein Problem, da diese in der Regel mit einem gewissen Vorlauf geplant werden. Staus hingegen können nicht mit Sicherheit vorhergesagt werden. Gleichzeitig sind sie nicht ortsgebunden, sondern bewegen sich und können sich ähnlich schnell auflösen wie sie entstanden sind. Daher tritt es regelmäßig auf, dass die an die Navigationssysteme gemeldeten Staus nicht mehr aktuell oder nicht aktuell genug sind. Für den Verbraucher kommt es dann zu dem unerfreulichen Fall, dass er im Stau steht, wo keiner gemeldet wurde, oder ihn das Navigationssystem einen Umweg um einen nicht vorhandenen Stau fahren lässt.

Ein Ansatz zur Lösung dieses Problems ist es, die Navigationssysteme selbst wiederum zu Sendern zu machen und sie ihre eigene Position übermitteln zu lassen. Genügend solche Positionsangaben vorausgesetzt können dann in Kombination mit der zeitlichen Entwicklung der einzelnen Positionen (Bewegung) und geeigneten Straßendaten Staus und sogar zäh fließender Verkehr in Echtzeit erkannt werden. Freilich sind dabei sehr große Datenmengen zu verarbeiten, was nur unter Ausnutzung von geeigneten Algorithmen und Datenstrukturen möglich ist.

Als Informatiker fragt man sich nun naturgemäß, wie solche Berechnungen effizient durchgeführt werden können und mit welchen Datenstrukturen gearbeitet werden muss. Dabei wird schnell klar, dass eine Vielzahl komplexer Probleme für die Berechnung von Routen oder auch nur für die Anzeige der Hintergrundkarte zu lösen ist. Eines dieser Probleme ist die Indizierung der Daten, die z.B. die Straßen oder Orte beschreiben. Diese allgemein als Geodaten bezeichneten Daten weisen durch ihre Geometrien leider Eigenschaften auf, die nicht sinnvoll von einem klassischen Index unterstützt werden können.

Gerade die schnelle Suche in sehr großen Beständen von Geodaten ist jedoch essenziell für jede komplexere Anwendung in diesem Bereich. Doch welche Indices soll man verwenden? Diese Fragestellung ergab sich auch für den Autor dieser Thesis bei seiner Tätigkeit bei einem Dienstleister für geographisches Datenmaterial, Standortanalysen und Immobilienbewertung[2]. Der erste Kontakt mit solchen Indices fand dabei mit den r-trees statt. Ein Kollege hingegen vertrat die These, das quadtrees das Mittel der Wahl seien. Nun stellte sich eine längere Phase der Koexistenz beider Indexfamilien ein bis im Zuge der Expansion ein weiterer Kollege hinzu kam, der von seinen positiven Erfahrungen mit

[2] on-geo GmbH, vormals inframation AG

kd-trees berichtete. Nun stellte sich für den Autor die Frage, welcher Index denn nun das Mittel der Wahl sei. Hieraus entwickelte sich dann auch letztlich das Thema dieser Thesis, mit der der Versuch gewagt werden soll, zumindest einige Hinweise für die Wahl des richtigen Index zu erhalten.

Doch bevor diese Frage geklärt werden kann, sind zuerst beginnend mit den Geodaten noch einige Grundlagen zu behandeln.

1.2 Geodaten

Um die Fragestellungen und Probleme näher verstehen zu können, die bei der Indizierung oder auch generell bei der Verarbeitung von geographischen Daten entstehen, istzuerst der Begriff der Geodaten zu definieren.

Vereinfacht ausgedrückt handelt es sich bei Geodaten um Daten, die einen räumlichen Bezug besitzen. Als Beispiel sei hier eine Straße genannt. Diese besitzt in der Regel einen Straßennamen, der für sich genommen nur eine Zeichenfolge ist. Verknüpft wird dieses Datum nun mit der Geometrie (einem Linienzug). Der räumliche Bezug entsteht, wenn diese Geometrie in einem geeigneten Koordinatensystem gespeichert wird, dass den „Vergleich“ mit anderen Straßenzügen erlaubt.

Bei dieser Schilderung handelt es sich allerdings, wie bereits gesagt, um eine vereinfachte Betrachtungsweise. Für das Beispiel der Straße ist bekannt, dass sie erst gebaut werden muss, bevor sie befahren werden kann. Diese Erkenntnis mag sich in diesem Kontext trivial anhören, jedoch handelt es sich dabei um eine weitere wichtige Eigenschaft von Geodaten, dem temporalen Bezug. Dabei besitzt ein „Objekt “ eine gewisse zeitliche Gültigkeit. Anzumerken ist, dass diese Eigenschaft in der Praxis zu weilen ignoriert oder zumindest nicht speziell betrachtet wird. Man geht dann einfach davon aus, dass ein kompletter Datensatz hinsichtlich seines temporalen Bezugs konsistent ist.

Als letzte ebenfalls wichtige, aber ebenso optionale Eigenschaft sind dann noch die topologischen Charakteristika zu nennen. Dahinter verbergen sich die Beziehungen zwischen den einzelnen „Objekten“. Als Beispiel wären hier zwei miteinander verbundene Straßen zu nennen. Die topologische Eigenschaft zwischen beiden Straßen ist dabei das „verbunden sein“. Zu beachten ist aber, dass diese Relationen losgelöst von den einzelnen Geometrien betrachtet werden muss! Die Eigenschaft „verbunden“ ist nicht ein Attribut einer oder beider Straßen. Vielmehr werden topologische Eigenschaften als vom konkreten Datensatz losgelöste Regeln oder Vorschriften betrachtet, welche auch nach Veränderungen an den Geometrien noch einzuhalten sind. Zur Verdeutlichung kann man sich z.B.

vorstellen, dass alle in einem Datenmenge erfassten Gebäude disjunkt sein müssen. Soll nun ein weiteres Gebäude hinzugefügt oder ein bestehendes verändert werden (Anbau), so ist dieses Charakteristikum weiterhin einzuhalten.

Zusammenfassend lassen sich die Eigenschaften in die folgenden Kategorien gliedern:[3]

Geometrische Eigenschaften Beschreiben die Form eines „Objekts“ sowie dessen genaue Position innerhalb eines Raums.

Topologische Eigenschaften Beschreiben die relativen räumlichen Beziehungen von „Objekten“ untereinander, ohne Berücksichtigung der konkreten Geometrien.

Temporale Eigenschaften Beschreibt die zeitliche Gültigkeit eines „Objekts“ sowie ggf. die zeitliche Veränderung seiner Eigenschaften (z.B. eine Bewegung eines Fahrzeugs oder ein Anbau eines Gebäudes).

Thematische Eigenschaften Beschreiben alle sonstigen Eigenschaften, die auch unter dem Begriff Sachattribute zusammengefasst werden können.

Ein einzelner Datensatz, also eine konkrete Ausprägung verschiedener Eigenschaften der genannten Kategorien, kann als „Objekt“ bezeichnet werden. Allerdings hat sich inzwischen auch durch die Standardisierung durch das Open Geospatial Consortium und die ISO-Spezifikation 19107[4] der Begriff des *Features* durchgesetzt. Im Rahmen der Thesis werden beide Begriffe synonym verwendet.

1.2.1 Simple Features

Wie leicht zu erkennen, ist ein Feature je nach Art seiner Eigenschaften relativ komplex in der Handhabung. Zudem werden in vielen Anwendungsfällen nicht alle Kategorien von Eigenschaften benötigt. Soll beispielsweise eine einfache Karte gezeichnet werden, so genügen einige thematische sowie eine geometrische Eigenschaft[5].

Zu diesem Zweck wurde durch das OGC der aus vier Teilen bestehende *Simple feature access* - Standard entwickelt. Dieser beschränkt die möglichen Eigenschaften eines Features genau auf die bereits genannten: einer geometrischen sowie beliebig vielen thematischen

[3] *Thomas Brinkhoff*: Geodatenbanksysteme in Theorie und Praxis, Heidelberg 2008, S. 60.

[4] ISO 19107:2003 Geographic information – Spatial schema, Techn. Ber., International Organization for Standardization (TC 211), 2003

[5] Sicherlich gibt es auch Anwendungsfälle, bei denen eine Karte mit Geodaten zu einem beliebig gewählten Stichtag erstellt werden soll. In der Regel ist aber das Kartenmaterial als ganzes hinsichtlich seiner temporalen Eigenschaften konsistent (statisch) und umfasst nur genau einen Stichtag. Als Beispiel seien hier die Kartendaten für Navigationssysteme genannt.

Eigenschaften. Darüber hinaus werden auch noch die möglichen Geometriearten eingeschränkt. Während ein normales Feature beliebige Geometrieinformationen z.B. Splines enthalten kann, erlaubt der Standard lediglich die folgenden sechs:

Point Ein einzelner Punkt.

LineString Ein aus n Punkten konstruierter Linienzug; die einzelnen Punkte sind durch Streckenabschnitte jeweils geradlinig miteinander verbunden.

Polygon Eine aus einem geschlossenen äußeren sowie beliebig vielen inneren geschlossenen Linienzügen (Ringen) konstruierte Fläche.

MultiPoint Eine Menge von *Points*.

MultiLineString Eine Menge von *LineStrings*.

MultiPolygon Eine Menge von *Polygonen*.

Zu beachten ist, dass abhängig vom Geometrietyp durch den Standard noch weitere Konstruktionsregeln existieren können, die jedoch wegen ihres Umfangs an dieser Stelle nicht weiter behandelt.

Nun mag man sich fragen, ob derartige Beschränkungen in der Praxis nicht zu zu großen Kompromissen bei der Datenerstellung führen. So lassen sich allein mit *LineStrings* keine echten Kurven modellieren. Hierzu ist festzustellen, dass für die meisten Anwendungsfälle die Approximation mittels eines *LineStrings* - genügend Punkte vorausgesetzt - vollständig ausreichend ist. Um dies zu verdeutlichen sei auf die Abbildung 1.1 verwiesen. Gezeigt werden darin am Beispiel einer Karte von Gummersbach die drei Geometrietypen, die an der blauen Hervorhebung zu erkennen sind. Konkret handelt es sich bei der Punktgeometrie um den Ortsnamen „Gummersbach“, beim LineString um die Bundesstraße 236 und beim Polygon um einen Wald. Insbesondere an den Kurven der B236 (und der anderen Straßen) ist zu erkennen, dass mittels Approximation ein akzeptables Ergebnis erzielt werden kann.

1.3 Geographische Informationssysteme

Um Geodaten erstellen und verarbeiten zu können, werden spezielle Anwendungen benötigt. Solche Anwendungen werden als Geographische Informationsysteme, Geoinformationssysteme oder kurz GIS bezeichnet. Sie existieren in unterschiedlichen Ausprägungen sowohl als komplett eigenständige Anwendung (z.B. Lösungen zur Erstellung/Anzeige

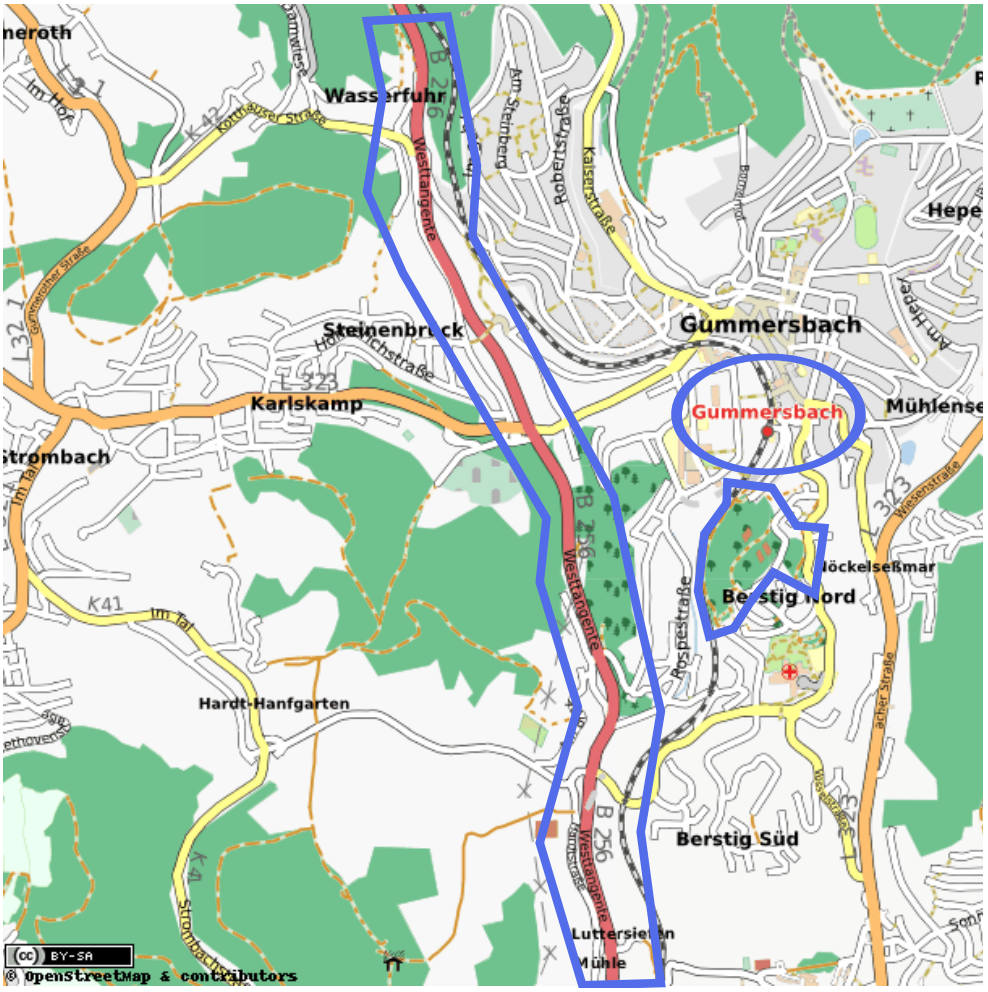

ABBILDUNG 1.1: Darstellung der Geometrietypen von Simple Features am Beispiel einer Karte von Gummersbach (Datenbasis: OpenStreetMap)

von Straßenkarten) oder als Teil eines größeren Anwendungssystems (z.B. Logistiksystemen mit integrierter Fahrzeugüberwachung). Unabhängig vom jeweiligen Einsatzzweck sind die meisten Geoinformationsysteme jedoch ähnlich aufgebaut. So lassen sie sich in vier voneinander unabhängige Komponenten gliedern:[6]

Erfassung Hierbei handelt es sich um die Eingabe-Schnittstelle. Die Erfassung kann über verschiedene Geräte (PC, GPS-Gerät) und Methoden (automatisiert oder manuell) erfolgen.

Verwaltung Diese Komponente übernimmt das Laden und Speichern von Geodaten in geeigneten Strukturen. Häufig wird dies inzwischen von speziellen Geodatenbanken übernommen.

[6] *Brinkhoff*: Geodatenbanksysteme in Theorie und Praxis (wie Anm. 3), S. 4.

Analyse inkl. Verarbeitung. Realisiert die Konvertierung, Transformation und Selektion sowie die räumliche Analyse und Interpolation von Daten. Zum Teil kann es zu Überschneidungen mit der Verwaltung, konkret mit den Geodatenbanksystemen kommen.

Präsentation In dieser Komponente wird die Ausgabefunktionalität realisiert. Diese kann sowohl graphisch oder multimedial gegenüber einem Benutzer oder auch als reiner Datensatz gegenüber einem anderen Geoinformationssystem erfolgen.

Auch wenn alle vier Komponenten gleichermaßen wichtig sind, soll, wie in der Einleitung angesprochen, im folgenden der Fokus auf der Verwaltungskomponente bzw. den Geodatenbanken liegen. Diese Datenbanksysteme müssen wegen der Komplexität von Geodaten besondere Anforderungen erfüllen. Insbesondere für die geometrischen Eigenschaften werden spezielle Lösungen benötigt. Man mag zu Anfang noch denken, dass die etablierten relationalen und objektrelationalen Datenbanksysteme ausreichen würden. Doch sei dazu nur auf die notwendigen (geometrischen) Funktionen verwiesen, die zur Selektion einzelner Features benötigt werden. Diese ließen sich zwar prinzipiell als *stored procedure* realisieren, jedoch ist die Verarbeitung von Millionen Geometrien zwecks Selektion einiger weniger für einen Kartenausschnitt mehr als unpraktikabel, zumal eine größere Anzahl an Anfragen pro Sekunde nicht ungewöhnlich ist.

Letztlich wird ein spezielles Datenbanksystem benötigt, was neben geometrischen Datentypen auch entsprechende geometrische und räumliche Operationen erlaubt. Gleichzeitig muss es in der Lage sein, diese Operationen über geeignete Indices möglichst effizient auszuführen. Zudem muss es zur Gewährleistung des räumlichen Bezugs auch entsprechende Koordinatensysteme beherrschen und idealerweise zwischen diesen transformieren können, um Abfragen über Geometrien unterschiedlicher Bezugssysteme zu unterstützen.

1.4 Zielsetzung

Im Rahmen dieser Thesis soll nun, wie bereits zu Anfang erwähnt, der Aspekt der Indices näher betrachtet werden. Dazu sollen die drei genannten Indexfamilien kd-tree, quadtree und r-tree miteinander verglichen werden. Zwar haben in den letzten Jahren im Bereich der Geodatenbanken die r-trees mit ihren Varianten eine größere Verbreitung gefunden. Dies ist aber keinesfalls derart ausgeprägt wie im Fall der B-Bäume in klassischen Datenbanksystemen.

Neben quadtrees, kd-trees und r-trees existiert freilich noch eine Vielzahl verschiedene Spezialindices, die in klar abgegrenzten Anwendungsszenarien signifikante Vorteile besitzen. Dies gilt beispielsweise für den M-tree[7] oder den Sa-tree[8]. Doch soll der Fokus hier allein auf den drei genannten liegen. In dieser Arbeit sollen alltägliche Situationen im Vordergrund stehen und auf die Betrachtung von Spezialfällen verzichtet werden.

Zielsetzung ist also ein Vergleich der drei Indexarten kd-tree, quadtree und r-tree. Dabei sollen exemplarisch je drei Varianten der jeweiligen Indices untersucht werden. Als Datenmaterial werden dazu sowohl statische als auch dynamische Geodaten herangezogen. Zudem sollen als Variante der dynamischen Daten auch „Bewegungsdaten“, also Geodaten, die einer kontinuierlichen Bewegung im Raum unterliegen (z.B. Fahrzeuge), Verwendung finden[9]. Analysiert werden sollen jeweils (einfache) gängige Abfragen, mit welchen ein Geoinformationssystem regelmäßig konfrontiert wird. Als Einschränkung sollen dabei jedoch (soweit möglich) keine Caches verwendet werden. Das Thema des Cachings ist zu umfangreich für eine angemessene Betrachtung im Rahmen der Thesis.

Neben dem reinen Vergleich auf Basis von Messdaten soll aber auch der grundsätzliche konzeptionelle Unterschied der verschiedenen Ansätze beleuchtet werden. Zweck hierbei ist es die verschiedenen Herangehensweisen an das Problem der Indizierung von Geometrien mit räumlichen Bezug aufzuzeigen und gegenüberzustellen.

Die Thesis gliedert sich dazu in fünf größere Abschnitte. Zuerst wird in Kapitel 2 die Testumgebung sowie das verwendete Datenmaterial vorgestellt. Anschließend stellen die Kapitel 3, 4 und 5 die drei Indexfamilien sowie ihre hier betrachteten Varianten vor. Im Kapitel 6 folgt dann der Vergleich der Indices. Den Abschluss bildet im Kapitel 7 ein Fazit zur gesamten Thesis.

[7] *Paolo Ciaccia/Marco Patella/Pavel Zezula*: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces, in: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97), San Francisco, CA, USA 1997, S. 426–435.

[8] *Gonzalo Navarro*: Searching in metric spaces by spatial approximation, in: The VLDB Journal 11 (1 2002), S. 28–46.

[9] Es wird ausdrücklich auf die Betrachtung der temporalen Eigenschaften der Bewegungsdaten verzichtet und ausschließlich die jeweils aktuelle Position indiziert.

Kapitel 2

Testumgebung

2.1 Anforderungen

Um den angestrebten Vergleich der Indices durchführen zu können, wurde eine entsprechende Testumgebung benötigt. Zweck dieser sollte es, wie bereits im Kapitel 1.4 geschrieben, sein, die verschiedenen Indices in unterschiedlichen Tests zu vergleichen. Ausgangspunkt der Entwicklung einer solchen Umgebung waren eine Reihe von Anforderungen, die zu Anfang formuliert wurden:

Modulare Tests Die einzelnen Tests sollen möglichst modular konstruiert sein. Ein neuer Test muss mit minimalen Änderungen am Hauptprogramm integriert werden können. Gleichzeitig sollen für einen Testlauf beliebige Tests miteinander kombinierbar (aneinanderreihbar) sein.

Einfache Integration der Indices Neue Indices sollen sich ohne größere Modifikationen in die Testumgebung einbauen lassen. Der konkret verwendete Index muss folglich gegenüber den Testfällen transparent sein.

Erfassung der Messwerte Die Testumgebung soll die Messwerte transparent erfassen. Zwecks Vergleichbarkeit darf dies nicht Aufgabe der Indices sein. Es müssen sowohl kontinuierlich während eines Testlaufs Werte erfasst werden können als auch Summen- und Durchschnittswerte am Ende eines solchen. Die Werte sollen automatisch geeignet gespeichert werden (CSV-Datei).

Datenspeicherung Die Indices sollen nicht direkt I/O-Operationen durchführen. Vielmehr soll dies über Pages geschehe. Ziel dieser Anforderung ist, die Erstellung neuer Indices erleichtern sowie auch eine bessere Vergleichbarkeit zu gewährleisten.

Testdaten Die für die Testläufe zu verwendenden Geodaten sollen nicht an konkrete Tests gekoppelt sein. Die Features sind vielmehr durch die Testumgebung in geeigneter gekapselter Form zu übergeben.

Format der Testdaten Die Testdaten sollen der Einfachheit halber als Shapefile[1] abgelegt werden. Da es sich dabei um den Quasi-Standard für den Datenaustausch im Bereich von Geoinformationssystemen handelt, wird hierdurch eine möglichst einfache Integration neuer Datenbestände gewährleistet. Das Testsystem muss dieses Format folglich unterstützen.

Bewegungsdaten Das Testsystem soll auf Basis eines Straßennetzes Bewegungsdaten von Kraftfahrzeugen generieren können. Die einzelnen Bewegungen sollen als diskrete Verschiebeoperationen (Startpunkt und Endpunkt der Bewegung während einer atomaren Zeiteinheit) an entsprechende Testfälle übertragen werden. Für Bewegungsdaten werden separate Testfälle benötigt.

Realisierungssprache Es soll Java als Realisierungssprache verwendet werden um einen möglichst einfache Portierbarkeit der Testumgebung insbesondere zwischen verschiedenen Betriebssystemen zu gewährleisten. Dies ist durch die dem Autor zur Verfügung stehenden Computersysteme zwingend notwendig.

Visualisierung Zur Analyse von Fehlern und zur Überprüfung der Implementierung der Indices soll die Testumgebung über eine Visualisierungsfunktion verfügen. Diese soll sowohl den Baum selbst als auch die von ihm erzeugte Unterteilung des Raums darstellen. Die erstellten Grafiken sollen als Vektordaten abgelegt werden.

2.2 Testdaten

Zur Durchführung der Testläufe wurde eine Menge von Testdaten benötigt. Diese sollten gleichzeitig auch die Datensätze für die Überprüfung der einzelnen Indeximplementierungen sein. Während ersteres einen großen realistisch aufgebauten Datensatz erfordert ist für letzteres eher ein kleiner überschaubarer Datensatz geeignet. Neben der Größe waren aber auch die Geometrien selbst von Bedeutung, genauer gesagt wurden verschiedene Arten von Geometrien benötigt. So sollten Punkt, LineString und Polygone für beide Größenklassen vorhanden sein. Daneben waren zur Erzeugung der Bewegungsdaten auch noch geeignete Straßennetze erforderlich.

Um diese Anforderungen erfüllen zu können und insbesondere auch realitätsnahe Geodaten zu verwenden, viel die Wahl grundsätzlich auf eine Untermenge der Daten aus dem

[1] *Environmental Systems Research Institute, Inc*: ESRI Shapefile Technical Description, Juli 1998.

OpenStreetMap-Projekt[2], konkret dem Land Nordrhein-Westfalen. Da es sich bei den Daten von OpenStreetMap um vektorbasiertes Kartenmaterial handelt ist die gewünschte Realitätsnähe gewährleistet. Zudem bietet die heterogene Struktur des Bundeslands Nordrhein-Westfalen mit dem dicht besiedelten Ruhrgebiet sowie einer Reihe dünnbesiedelter Regionen eine relativ gute Mischung verschiedener Anwendungsfälle. Außerdem existieren Geometrien in allen Varianten und es können zudem auch Straßennetze aus den Daten abgeleitet werden.

Leider ist es nicht möglich direkt die gewählten Daten von OpenStreetMap in Shapefiles exportieren zu lassen. Es waren daher zusätzliche Tools notwendig um die entsprechenden Konvertierungsschritte durchzuführen. Daneben werden auch ausgewählte Regionen von externen Anbietern als Shapefiles angeboten.

Um neben den eigentlichen Testläufen auch Überprüfungen vornehmen zu können, sind, wie bereits erwähnt, kleinere Datenbestände erforderlich. Hierfür wurden zwei Untermengen des gewählten OpenStreetMap-Datensatzes gewählt. Zum einen der Campus Gummersbach der Fachhochschule Köln sowie Gummersbach mit Umgebung. Ersterer ist in Abbildung 2.1, letzterer in Abbildung 2.2 zu sehen. Die genaue Zusammensetzung der Datenbestände hinsichtlich der Punkte, LineStrings, Polygons sowie Straßen (Straßenabschnitte) wiederum zeigt Tabelle 2.1.

Abbildung 2.1: Darstellung des gewählten kleinräumigen Datenbestands Campus Gummersbach (Datenbasis: OpenStreetMap)

[2]http://www.openstreetmap.de

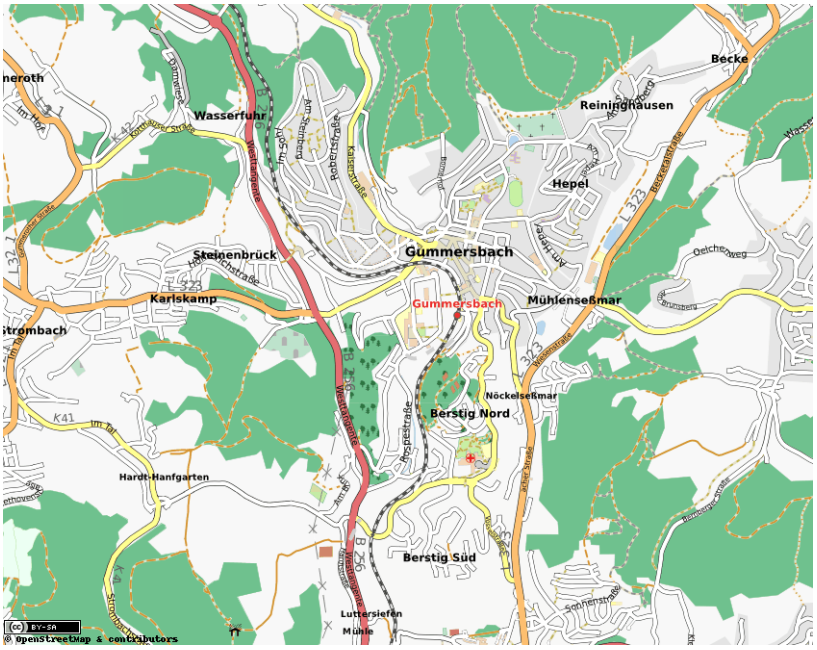

Abbildung 2.2: Darstellung des gewählten kleinräumigen Datenbestands Gummersbach und Umgebung (Datenbasis: OpenStreetMap)

	Campus Gummersbach	**Gummersbach und Umgebung**
Points	14	737
LineStrings	42	918
Polygons	16	216
Straßen	13	509

Tabelle 2.1: Zusammensetzung der Datenbestände Campus Gummersbach sowie Gummersbach und Umgebung

Anzumerken ist, dass beide kleinräumigen Datensätze direkt von OpenStreetMap bezogen und über entsprechende Tools konvertiert wurden. Der Komplettbestand NRW hingegen stammt von einem externen Anbieter. Hierdurch ergibt sich eine andere (feinere) Aufteilung als die beiden kleineren Beständen. Dabei wird nicht nur in die reinen Geometrietypen sowie die Straßen gegliedert, sondern entsprechend logische Kategorien wie Wege, Orte, Natur und Gebäude. Eine solche Aufteilung kann unter dem Gesichtspunkt der Praxisnähe als realistischer eingestuft werden. Dies liegt zum einen in der erleichterten Wartung. Zum anderen ermöglicht dies gezielte inhaltliche Abfragen wie „Wo ist die nächste Tankstelle“ oder „Wie groß ist der Wald bei Position x/y“ ohne alle gleichartigen Geometrien ebenfalls analysieren zu müssen oder den Index auf thematische Attribute auszuweiten. Die genaue Unterteilung des Datenbestands zusammen mit den jeweiligen Anzahlen und Geometrietypen kann aus Tabelle 2.2 entnommen werden.

	Typ	**Anzahl**
Gebäude	Polygon	827335
Natur	Polygon	62009
Orte	Punkt	8319
Points (of Interest)	Punkt	221608
Eisenbahntrassen	LineString	34696
Fuß- und Fahrwege	LineString	910485
Wasserwege	LineString	24397

TABELLE 2.2: Zusammensetzung des Datenbestands Nordrhein-Westfalen

2.3 Realisierung

Um die Anforderungen umsetzen zu können wurden zwei separate Anwendungen erstellt. Die erste Anwendung dient zum Ausführen von Testläufen, die Aufgabe der zweiten ist die Visualisierung der erstellten Bäume. Beide Anwendungen fungieren dabei jedoch nur als Frontend und nutzen letztlich die gleiche Codebasis.

Diese wiederum ist neben den Anwendungen in fünf Komponenten gegliedert, die sich unmittelbar aus den Anforderungen ergeben. Dabei handelt es sich um die Folgenden:

- Pageverwaltung
- Index-Interface
- Indexvisualisierung
- Test-Interface
- Generator für Bewegungsdaten

Das genaue Zusammenspiel der verschiedenen Komponenten sowie der beiden Anwendungen wird im Abbildung 2.3 gezeigt. Die Pfeilrichtungen symbolisieren jeweils die Aufrufrichtungen. Ausgangspunkt ist jeweils die konkrete Anwendung. Im Falle der Testläufe startet diese das Test-Interface mit der Information, welcher Test und welcher Index zu verwenden ist. Dieses wiederum instantiiert den jeweiligen Index und verknüpft ihn mit der Pageverwaltung. Dabei wird von Seiten des Index die gewünschte Pagegröße gemeldet[3]. Zum Ausführungszeitpunkt des Testfalls können dann durch diesen beliebig viele Operationen auf dem Index durchgeführt werden. Zu beachten ist die Aufrufrichtung

[3]Praktisch gesehen könnte die Pageverwaltung eine eigene, feste Pagegröße verwenden. In diesem Fall müsste ein Mapping auf die durch den Index benötigten Größen stattfinden.

des Generators für Bewegungsdaten. Dieser ruft den aktiven Testfall auf, welcher als Listener fungiert. Die Visualisierungsanwendung hingegen kommuniziert mit der Index-Visualisierung. Dabei wird der zu verwendende Index übertragen. Diese wiederum startet genauso wie während Testläufen auch das Index-Interface.

Zur besseren Automatisierung der Testläufe wurden zudem Annotationen für Indices und Tests formuliert, die die jeweiligen Anforderungen und Fähigkeiten beschreiben. Als Beispiel sei die Fähigkeit genannt Nicht-Punktgeometrien zu indizieren oder die Anforderung seitens der Tests dynamische Daten verarbeiten zu können. Die Anwendung kann so sicherstellen, dass bei einer größeren Menge an Tests und Indices nur diejenigen kombiniert und ausgeführt werden, die auch zu einander sowie zum ausgewählten Datensatz passen.

Anzumerken ist, dass die Generierung der Bewegungsdaten eine gewisse Sonderstellung einnimmt und deshalb einer separaten Betrachtung bedarf. Genauere Informationen hierzu werden im Abschnitt 2.3.1 dargelegt. Ebenfalls eines Hinweises bedarf die Realisierung der Shapefile-Unterstützung. Diese wurde mittels des Geotools-Framework[4] realisiert. Der positive Nebeneffekt dabei ist, dass prinzipiell auch andere Datenquellen genutzt werden können, da das Framework lediglich ein allgemeines Interface für Datenzugriffe der jeweiligen Anwendung zur Verfügung stellt.

Bezüglich des Index-Interfaces ist noch zu erwähnen, dass alle grundlegenden Such- und Manipulationsoperationen in jedem Index separat implementiert werden müssen. Wegen der Verschiedenartigkeit der zu vergleichenden Indices ist hier keine praktikable Universallösung möglich. Im Falle von Suchoperationen wird auch auf den durchaus denkbaren Fallback einer Durchsuchung aller eingefügten Features verzichtet, da dies im Rahmen des angestrebten Vergleichs keinen sinnvollen Zweck erfüllen würde. Gleichwohl wäre in der Praxis ein solches Vorgehen möglich.

Die Wahl dieser Vorgehensweise ist jedoch auch mit einem gewissen Nachteil verbunden. So wurden letztlich nur solche Operationen realisiert, die auch tatsächlich von Seiten der Testfälle benötigt wurden. Die Implementierung zusätzlicher Methoden wäre zwar für praktische Anwendungen hilfreich gewesen, hätte aber den Zeitrahmen gesprengt. Eine Übersicht über die tatsächlich unterstützten Operationen gibt die Tabelle 2.3.

Wie vermutlich bereits aufgefallen, wurde in den bisherigen Betrachtungen die Sammlung der Messwerte nicht berücksichtigt. Dies liegt darin begründet, dass es sich hierbei um eine Querschnittaufgabe handelt, sie folglich von allen bzw. den meisten Komponenten realisiert werden muss. Die genaue Umsetzung erfolgte dabei weitestgehend mit Hilfe von Delegates, welche jeweils die einzelnen Methodenaufrufe für die konkreten Messungen erfassen. Ein Vorteil der Delegates ist, dass die Erfassung der Messwerte nicht selbst

[4] http://geotools.org

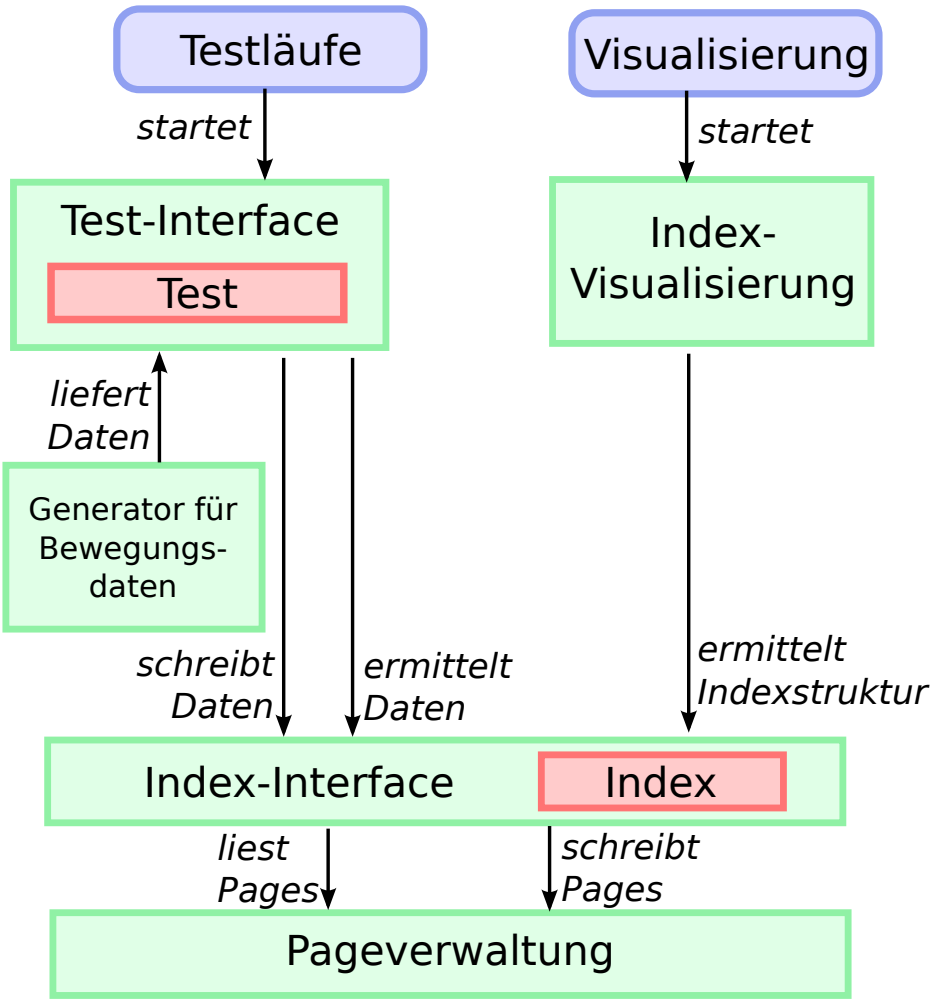

ABBILDUNG 2.3: Darstellung der einzelnen Komponenten der Testumgebung sowie ihres Zusammenspiels

zum Teil der Messung wird. Zudem verhindert dieses Vorgehen, dass die Funktionalität wiederholt implementiert werden muss.

In Tabelle 2.4 sind alle erfassten Messwerte zusammen mit dem Komponente aufgeführt, innerhalb der sie ermittelt werden. Die Sammlung der durch die Delegates erfassten Werte liegt im Aufgabenbereich der Anwendung. Dies kann sowohl kontinuierlich, also nach jedem einzelnen Test (typischerweise Identisch mit einer Operation auf dem Index) oder nach Abschluss einer Anzahl von Testläufen erfolgen. Da einige Messwerte nur auf Anfrage erfasst werden (z.B. die Baumtiefe) hat dies auch einen gewissen Einfluss auf den Umfang der Messungen und die damit verbundene Gesamtlaufzeit.

Zu dem Messwerten ist noch anzumerken, dass auch der tatsächliche Umfang an gelesenen oder geschriebenen Bytes bestimmt werden kann. Genauer gesagt handelt es sich dabei um abgeleitete Werte, die aus der Multiplikation von Pagegröße und der Anzahl der

Operation	Beschreibung
`buildIndex`	Erstellt einen Index aus einer Menge von Features (statische Datenmenge)
`add`	Fügt ein einzelnes Feature zum Index hinzu
`remove`	Entfernt ein einzelnes Feature aus dem Index
`move`	Verschiebt ein Feature (Punktgeometrie) zu einer anderen Position (diskrete Bewegung)
`bbox`	Selektiert alle Features in einem bestimmten Bereich (Rechteck, auch als Bounding Box bzw. BBox bekannt)
`location`	Selektiert alle Features an einer bestimmten Position
`nearestNeighbor`	Findet die `n` nächsten Nachbarn zu einem gegebenen Punkt

Tabelle 2.3: Auflistung aller durch das Index-Interface unterstützten Such- und Manipulationsoperationen

Messwert	Einheit	Komponente
Pagegröße	Byte	Pageverwaltung
Gelesene Pages	Pages	Pageverwaltung
Geschriebene Pages	Pages	Pageverwaltung
Gelöschte Pages	Pages	Pageverwaltung
Page-Splits	Pages	Pageverwaltung
Gesamtanzahl Pages	Pages	Pageverwaltung
Baumtiefe	Knoten	Index-Interface
Durchschnittliche Knotenauslastung	Prozent	Index-Interface
Gesamtdauer alle Operationen	ms	Index-Interface
Gesamtdauer Einfügeoperationen	ms	Index-Interface
Gesamtdauer Suchoperatione	ms	Index-Interface
Gesamtdauer Löschoperationen	ms	Index-Interface
Gesamtdauer Verschiebeoperationen	ms	Index-Interface
Gesamtdauer Nearest-Neighbor-Operationen	ms	Index-Interface

Tabelle 2.4: Durch die Testumgebung erfasste Messwerte

gelesenen oder geschriebenen Pages ermittelt werden können. Eine separate Messung dieser Werte durch die Delegates ist folglich nicht notwendig.

2.3.1 Generator für Bewegungsdaten

Während die anderen Komponenten der Testumgebung jeweils für die Thesis neu entwickelt wurden, ist beim Generator für Bewegungsdaten ein anderes Vorgehen gewählt worden. Der Grund hierfür ist in der signifikanten Komplexität des Themas zu suchen. Da im Rahmen dieser Thesis jedoch nur die Ergebnisse eines solchen Generators benötigt wurden, schien es nicht sinnvoll hier eine eigene Lösung zu entwickeln und anschließend zu implementieren. Zudem würde dies auch deutlich über den Umfang dieser Arbeit hinausgehen.

Stattdessen wurde der von Prof. Dr. Thomas Brinkhoff in Java entwickelte „Network-based Generator of Moving Objects“[5] verwendet. Zu dem Generator existiert auch eine theoretische Beschreibung der zu Grunde liegenden Konzepte sowie der angewandten Vorgehensweise[6]. Der Generator ist in der Lage die Bewegungen von Punkt- und Flächengeometrien in einem gegebenen Netzwerk zu simulieren. Die Anzahl der Objekte sowie die zeitliche Entwicklung eben dieser Anzahl kann über entsprechende Parameter festgelegt werden. Abbildung 2.4 zeigt die Oberfläche des Generators mit dem Straßennetz des Datensatz „Gummersbach und Umgebung“.

Um den Generator in die Testumgebung zu integrieren wurden einige Anpassungen durchgeführt. Dieser Schritt war notwendig, da dieser ursprünglich nur als eigenständiges Programm konzipiert wurde. So ist nun die direkte Anbindung von Shapefiles mittels Geotools möglich. Zudem kann der Generator jetzt ohne Oberfläche ausgeführt werden. Die Rückmeldung der jeweiligen Bewegungen im Netzwerk erfolgt dann über einen Listener, welcher nur noch Kombinationen von Objekt-ID sowie Start- und/oder Zielposition gemeldet bekommt.

2.4 Testfälle

Es wurden insgesamt acht Testfälle erstellt, die in drei unterschiedliche Gruppen eingeordnet werden können. Bei den drei Gruppen handelt es sich um Tests mit statischen Daten,

[5] Der Generator kann über die Homepage von Prof. Dr. Thomas Brinkhoff bezogen werden: http://iapg.jade-hs.de/personen/brinkhoff/generator/

[6] *Thomas Brinkhoff*: A Framework for Generating Network-Based Moving Objects, in: Geoinformatica 6 (2 2002), S. 153–180.

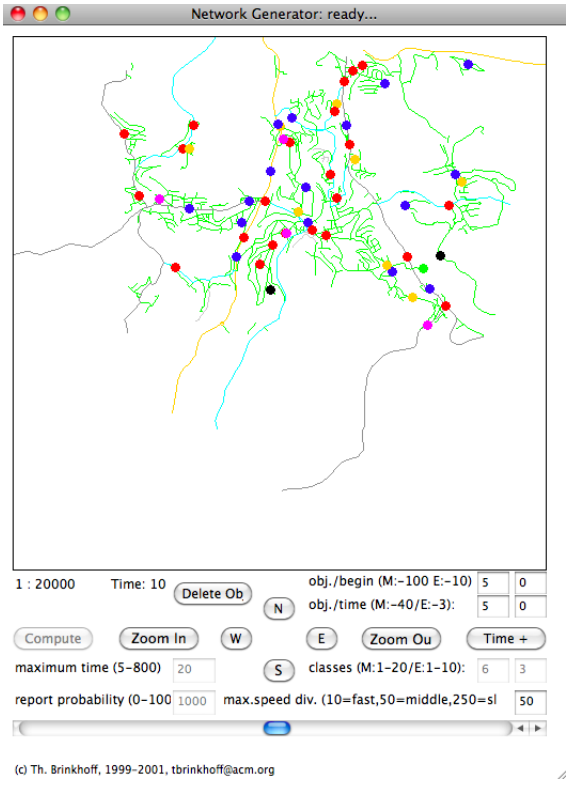

ABBILDUNG 2.4: Darstellung der GUI des *Network-based Generator of Moving Objects* mit den Straßennetz des Datensatz „Gummersbach und Umgebung“

mit dynamischen Daten sowie schlussendlich solchen mit Bewegungsdaten[7]. Die Tests sind in dabei aus einer oder eine Menge von Einzeloperationen aufgebaut, die entweder Selektions- oder Manipulationsoperationen sein können.

Die Selektionsoperationen verwenden in der Regel zufällige Geometrien um eine entsprechende Abfrage durchzuführen. Dabei wird sichergestellt, dass die Geometrie innerhalb der Grenzen des Datensatzes liegt. Es kommt somit nicht vor, dass bei Nutzung eines kleinräumigen Datensatzes wie des Campus Gummersbach Abfragen in Dortmund oder Köln durchgeführt werden.

Manipulationsoperationen hingegen verwenden keine vollständig zufällig generierten Datensätze. Hier werden vielmehr Features direkt aus dem gewählten Datensatz oder, im Falle von Bewegungsdaten, hiervon abgeleitete Daten, verwendet. Somit liefern Selektionsoperationen auch bei der Kombination mit Manipulationsoperationen (dynamische Daten) immer eine Untermenge des verwendeten Datensatzes als Ergebnis. Gleichwohl geschieht die Auswahl eines zu verändernden Features oder die Festlegung einer Route eines Objekts in der Bewegungssimulation wiederum zufällig.

[7] Bewegungsdaten sind letztlich auch nur dynamische Daten mit einer zusätzlichen zeitlichen Komponente. Da ihnen hier jedoch eine besondere Bedeutung zukommt, werden sie als separate Gruppe aufgeführt.

2.4.1 Selektionsoperationen

An dieser Stelle ist es sinnvoll kurz auf die verschiedenen möglichen Selektionsoperationen einzugehen. Die beiden am häufigsten eingesetzten und damit auch die bekanntesten sind die Bounding Box-Abfrage, auch Window-Query genannt sowie die Positionsabfrage (Location Query). Bei ersterer wird ein Rechteck gewählt. Anschließend werden all jene Geometrien selektiert, die ganz oder teilweise innerhalb des Rechtecks liegen. Man kann also davon sprechen, dass diese das Abfragerechteck schneiden, weshalb diese Abfrage auch als *Intersection*-Query bezeichnet werden kann.

Es ist jedoch dabei eine gewisse Vorsicht geboten, da abhängig von der Anwendungsschicht, an die die Abfrage gerichtet ist, auch etwas anderes gemeint sein kann. Befragt man nämlich die Schnittstelle eines GIS, so erhält man hier ein anderes Ergebnis als wenn ein Index direkt befragt wird. Indices arbeiten - wie in den nachfolgenden Kapiteln näher erläutert - häufig auf Basis von Minimum Bounding Rectangles (MBRs) um die Geometrien der Features zu indizieren[8]. Wird nun eine entsprechende Abfrage durchgeführt, ermittelt ein Index folglich das Ergebnis auch nur auf Basis dieser kleinst möglichen Rechtecke um eine Geometrie herum. Es werden folglich in der Regel nur Kandidaten zurückgegeben[9]. Ein GIS führt anschließend die gewählte Selektionsoperation auf den echten Geometrien durch. Der Unterschied zwischen der Bounding Box- sowie der *Intersection*-Query aus Sicht des GIS besteht nun darin, dass bei ersterer eben jene „Nachbearbeitung“ der Kandidaten entfällt. Prominenter Anwendungszweck einer solchen Window Query ist die Kartendarstellung, bei der alle Features im Kartenausschnitt ermittelt werden sollen. Hier ist es im Allgemeinen nicht von Bedeutung, ob zu viele Features (Kandidaten) ermittelt werden, da diese während des Zeichenvorgangs der Karte verworfen werden können[10].

Bei der Positionsabfrage bzw. Location Query werden all jene Geometrien ermittelt, die einen bestimmten Punkt schneiden. Die Abfrage ist folglich vergleichbar zur BBox-Query, jedoch mit einer anderen Suchgeometrie. Auch hier ist aus Sicht eines GIS zwischen Kandidaten und echtem Ergebnis zu unterscheiden. Diese Art von Abfrage findet immer dann statt, wenn bestimmte Gebiete an einer gegebenen Position ermittelt werden sollen, z.B. das Postleitzahlgebiet oder die Gemeinde. Eine Anwendung auf Punkt- oder LineString-Datensätze ist unüblich. Nun mag sich der geneigte Leser fragen, warum zwischen beide Operationen unterschieden werden muss, da ja prinzipiell die BBox-Query auch mit der

[8] Neben MBRs, also Rechtecken, sind noch weitere Approximationen der eigentlichen Geometrie denkbar und finden auch in einigen Indices Anwendung, so z.B. Dreiecke

[9] Im Falle von einzelnen Punktgeometrien handelt es sich bereits um die echten Ergebnisse, da der MBR eines Punktes identisch zum Punkt selbst ist

[10] Der Render-Code führt somit seine eigene Überprüfung der Geometrien durch. Dies stellt in der Praxis normalerweise keinen zusätzlichen Aufwand dar, da ein solcher Schritt in Folge von Multi-Geometrien sowie der in der Regel durchzuführenden Projektion sowieso erfolgen muss.

Bounding Box eines Punktes durchgeführt werden kann. Dazu sei angemerkt, dass im Falle von Location Queries bei vielen Indices günstigere Optimierungen durchgeführt werden können, vor allen dann, wenn bereit vorab bereits klar ist, dass bei Abfrage mit einem Punkt nur ein einziger Pfad durchlaufen werden muss. Eine solche Optimierung ließe sich zwar auch in die BBox-Query integrieren, jedoch wäre auch dann ein separater Test angebracht.

Neben diesen beiden einfachen gibt es noch einige komplexere Relationsoperationen, beispielsweise: *disjoint*, *contains* oder *overlap*. Eine vollständige Auflistung wird in Abbildung 2.5 gezeigt. Auffällig mag bei dieser Grafik sein, dass die bereits erwähnte *intersects*-Operation nicht vorkommt. Doch hier trügt der erste Anschein, da ein *intersects* letztlich nur die Negation eines *disjoint* ist. Somit ist diese Operation gleich fünf Mal Bestandteil der Abbildung!

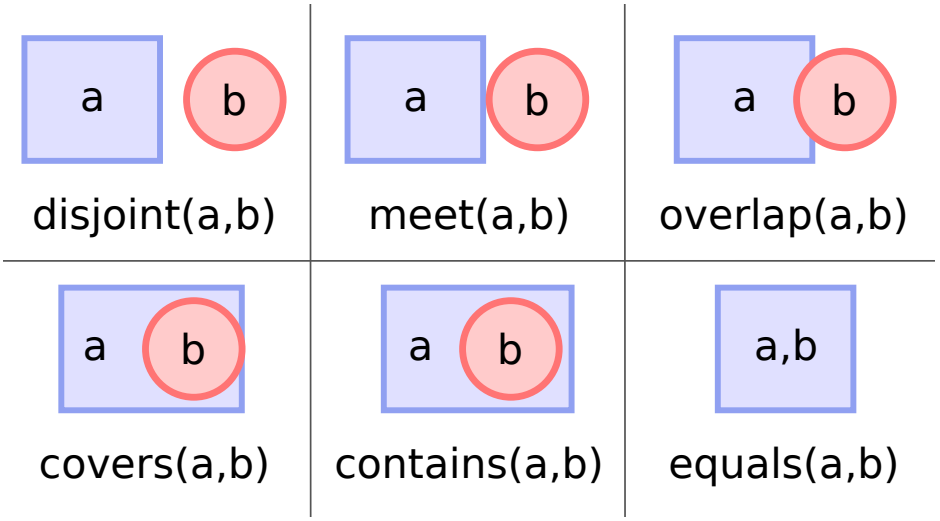

Abbildung 2.5: Auflistung der sechs bekannten Relationsoperationen

Die genannten räumlichen Abfragen lassen sich ähnlich zu den beiden Eingangs erwähnten zwecks Nutzung eines Index auf MBRs anwenden. Das Verfahren ist dabei jeweils vergleichbar und unterscheiden sich im wesentlichen durch die Anzahl der zu durchlaufenden Zweige im Baum (festgelegt durch die Auswahlbedingung). So können im Falle von *disjoint*, also der Suche nach allen zu einem gegebenen Rechteck disjunkten MBRs, nur sehr wenige Zweige des Baums verworfen werden, was zu einer relativ hohen Laufzeit führt.[11].

Letztlich wurde, wie bereits in Tabelle 2.3 erkennbar, auf die Implementierung und damit auch auf den Test von solchen weitergehenden Realtionsoperationen verzichtet. Hierfür sind zwei Gründe zu nennen. Erstens werden die beiden grundlegenden Abfragen der

[11] *Yannis Manolopoulos* u. a.: R-Trees: Theory and Applications (Advanced Information and Knowledge Processing), 1. Aufl., Sep. 2005, S. 55.

Window Query und *Location Query* in der Praxis sehr viel häufiger angewendet. Zweiten unterscheiden sie sich bei den hier gewählten Indices und deren Varianten auch nur durch das Auswahlkriterium der jeweils zu durchlaufenden Teilbäume von einander.

Neben den Relationsoperationen zwischen zwei Geometrien gibt es noch eine weitere Gruppe von Selektionsoperationen: der Suche nach den nächsten Nachbarn (Nearest Neighbor Query). Hier wird zu einer gegebenen Geometrie `P` eine konfigurierbare Anzahl `n` an nächstgelegenen Features ermittelt. Alle `n` Features liegen dabei näher an `P` als irgendein anderes Feature der Datenmenge. Neben dieser klassischen Version gibt es noch eine Reihe von Varianten der Nearest Neighbor Query, z.B. die Suche der nächsten Nachbarn mit bestimmten Attributen oder in bestimmten Himmelsrichtungen. Im Rahmen dieser Thesis wurde jedoch nur grundlegende Version realisiert.

2.4.2 Statische Geodaten

Nach der Sensibilisierung für die verschiedenen Selektionsoperationen ist es nun angebracht die einzelnen Tests der genannten drei Gruppen vorzustellen, insbesondere der jeweils durch diese ausgeführten Operationen.

Zur ersten Gruppe, die die Tests auf Basis von statischen Geodaten umfasst, können vier Testfälle gezählt werden:

BuildIndexTest Erstellt eine statischen Index aus einer gegebenen Menge von Geodaten

BBoxQueryTest Erzeugt pro Lauf eine Abfrage auf einem statischen Index mit zufällig erzeugten Bounding Boxes. Es ist dabei gewährleistet, dass die Bounding Boxes jeweils vollständig innerhalb der Gesamtfläche des Datensatzes liegen.

LocationQueryTest Erzeugt pro Lauf eine Abfrage auf einem statischen Index nach Objekten an einer zufällig gewählten Position. Genauso wie beim *BBoxQueryTest* ist dabei gewährleistet, dass der Abfragepunkt nur innerhalb der Gesamtfläche des Datensatzes liegt.

NearestNeighborTest Vergleichbar zum *LocationQueryTest*, jedoch wird in diesem Fall jeweils der nächste Nachbar gesucht

2.4.3 Dynamische Geodaten

Zur Gruppe der Tests mit dynamischen Daten kann direkt nur ein einziger gezählt werden. Dies mag sich nach relativ wenig anhören, doch wird durch diesen Test bereits ein relativ

großes Spektrum an Operationen abgedeckt. Zudem existieren mit den Tests zu Bewegungsdaten noch weitere wenn auch spezialisiertere. Der Testfall, genannt *ReadWriteTest* wählt in jedem Schritt eine Operation aus. Welche wird dabei durch eine prozentuale Wahrscheinlichkeitsangabe festgelegt. Als Einzeloperationen gibt es das Hinzufügen eines Features, das Löschen eines Features, die Bounding Box-Abfrage mit einem zufälligen Auswahlbereich sowie die Abfrage des nächsten Nachbarn zu einer zufällig gewählten Position. Auf die Einbeziehung der Location Query wird verzichtet, da der Ziel dieses Tests, die Degenerierung von Bäumen in Folge von Manipulationsoperationen zu untersuchen, bereits sehr gut mit der Window Query abgedeckt ist. Die grundsätzlichen Laufzeitunterschiede zwischen Window Query und Location Query hingegen können besser bei entsprechenden Einzeltests analysiert werden.

Durch die zufällige Wahl bestimmter Einzeloperationen wird letztlich versucht, ein reales Nutzungsverhalten mit Einfüge-, Lösch- und Abfrageoperationen zu simulieren. Dazu wurde die folgende Verteilung der Operationen gewählt: 20% Einfüge-, 10% Lösch-, 20% Nearest Neighbor- und 50% Bounding Box-Operationen. Somit sind 30% der Operationen Manipulationsoperationen. Dieser Wert mag vielleicht etwas hoch gewählt erscheinen, jedoch dient diese Maßnahme dazu die Gesamtlaufzeit des Testes im vertretbaren Rahmen zu halten.

Bis zu diesem Zeitpunkt nicht berücksichtigt wurde die Frage, wie der Index beim Start der Tests aufgebaut ist. Sicherlich ist leicht ersichtlich, dass ein initial leere Index gerade bei den ersten Operationen zu nicht unbedingt repräsentativen Ergebnissen führen würde. Daher werden zu Anfang 20% der im Datensatz enthaltenen Features zufällig ausgewählt und mittels einzelner Einfügeoperationen in den Index integriert.

2.4.4 Bewegungsdaten

Im Kontext der Bewegungsdaten wurden drei unterschiedliche Tests realisiert. Der erste, *MovingObjectsTest* simuliert dabei die reinen Bewegungsoperationen auf dem Index. Damit können mittels dieses Testfalls auch nur die entsprechenden Manipulationsoperationen verglichen werden und ist dementsprechend eher von geringem Interesse. Aus diesem Grund wird er auch für den weiteren Vergleich nicht herangezogen. Er bildet allerdings zugleich auch die Basis für die beiden anderen Testfälle: *MovingObjectsWithBBoxTest* und *MovingObjectsWithNearestNeighborTest*. Ersterer führt zusätzlich nach einer bestimmten Anzahl an Fahrzeugbewegungen eine Window Query mit einer entsprechend zufällig bestimmten Bounding Box durch. Letzter arbeitet ähnlich, nur dass stattdessen eine Abfrage des nächsten Nachbarn zu einem zufällig bestimmten Punkt durchgeführt

wird. Bei beiden wird standardmäßig alle drei Bewegungsoperationen eine entsprechende Selektionsoperation durchgeführt.

2.4.5 Validierung

Ein Teil der Testfälle ist, wie bereits angedeutet, auch in der Lage die jeweilige Indeximplementierung hinsichtlich ihrer Funktionalität zu überprüfen. Getestet wird dabei mittels geeigneten Selektionsoperationen und anschließendem Vergleich gegen die erwartete Menge. Diese wiederum wird eine entweder mittels Geotools oder, falls davon nicht unterstützt, durch Anwendung der Selektionsbedingung auf alle eingefügten Features ermittelt.

Im Falle von Manipulationsoperationen kann entweder durch Auslesen aller Features mit einer geeignet großen Bounding Box-Selektion die reine Existenz eines Features im Index überprüft werden. Oder es kann mittels kleinerer Bounding Box-Selektion um das manipulierte Feature herum eine relativ kleine Menge ausgelesen werden. Hierdurch wird mit einiger Gewissheit die Existenz eines bestimmten Features an einer bestimmten geographischen Position im Index überprüft. Letztlich kommen in den Testfällen beide Varianten zum Einsatz, abhängig von der entsprechenden Manipulationsoperation.

Anzumerken ist, dass die Validierung im Rahmen Testfälle nicht den Anspruch hat, alle möglichen Randbedingungen und Codepfade zu überprüfen. Vielmehr dienen sie dazu grobe Implementierungsfehler einfach relativ einfach entdecken zu können. Sie ist also als ergänzende Funktion zur Visualisierungsapplikation und zu sonstigen Überprüfungen zu verstehen.

Kapitel 3

kd-trees

3.1 Einführung

Mit der bekannteste Index überhaupt dürfte der klassische binäre Suchbaum sein. Jedem Knoten ist dabei ein eingefügter Wert zugewiesen und jeder hat maximal zwei Kindknoten. Diese wiederum sind für einen Knoten so definiert, dass das linke nur Werte kleiner als der eigene und rechte nur Werte größer oder gleich dem eigenen Wert besitzen kann[1].

Angenommen man ist nun mit einer relativ großen Menge an Features konfrontiert, deren Geometrie immer nur aus exakt einem Punkt besteht. Nun sollen genau diese Geometrien indiziert werden um beispielsweise eine Bounding Box-Abfrage zu beschleunigen. Eine Möglichkeit wäre einen binären Suchbaum zu verwenden. Da es sich um einen relativ einfachen Baum handelt ließe sich eine Implementierung entsprechend schnell erstellen. Die Verknüpfung zwischen Geometrie im jeweiligen Knoten und Feature wäre dann jeweils über eine einfache Referenz realisiert.

Leider zeigt sich genau bei der Geometrie schnell ein grundlegendes Problem: Punktgeometrien besitzen im zweidimensionalen Raum genau zwei Attribute (Dimensionen). Mit einem binären Suchbaum kann ohne weiteres jedoch nur ein einzelnes Attribut indiziert werden. Folglich wäre man gezwungen sich für den x- oder y-Wert zu entscheiden. Abbildung 3.1 zeigt beispielhaft einen solchen binären Suchbaum, bei dem der x-Wert als Kriterium gewählt wurde[2]. Der y-Wert ist somit nur über das Feature selbst zugänglich (in der Abbildung der Einfachheit halber in Klammern dargestellt). Eine Suchoperation

[1] Es ist genauso möglich das linke Kind als kleiner **oder** gleich und das rechte entsprechend nur als größer definiert. Prinzipiell funktionieren beide Varianten gleich gut, im Kontext der kd-trees hat sich jedoch die im Text genannte Betrachtungsweise durchgesetzt.

[2] Die Reihenfolge, in der die Features eingefügt wurden, lautet: 25/10, 15/75, 15/9, 24/57, 30/25, 29/29, 95/11

könnte daher nur auf dem x-Wert realisiert werden. Dies wiederum hat zur Folge, dass lediglich Kandidaten mittels Index ermittelt werden können, die anschließend in einem zweiten Schritt bzgl. des y-Wertes zu überprüfen sind. Bereits hier wird also deutlich, dass das Vorgehen ein gewisses Problem mit sich bringt.

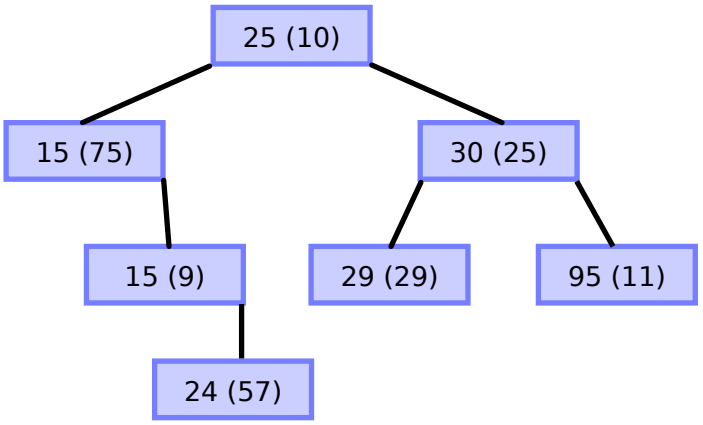

Abbildung 3.1: Binärer Suchbaum für Punktgeometrien. Kriterium ist nur der x-Wert.

Es gibt jedoch auch bei der Suche selbst eine Schwierigkeit. So sind Geodaten nur selten gleichmäßig verteilt. Häufig konzentrieren sie sich vielmehr in bestimmten Regionen (Ballungsräume). Wird nun beispielsweise in einem Datensatz der Einzelhändler in ganz Deutschland gesucht und angenommen die entsprechende Bounding Box entspräche einem Ausschnitt der Küste von Mecklenburg-Vorpommern, so kann es zu einer sehr großen Anzahl an Kandidaten kommen. Mag man sich zuerst darüber wundern, so wird in Abbildung 3.2 schnell die Ursache hierfür klar. Angenommen der Blau markierte Bereich sei die Bounding Box, so würden auch alle Features im rot markierte Bereich zu Kandidaten. Wie in der Abbildung zu sehen ist, wäre somit auch der sehr große Ballungsraum Berlin dabei einbezogen worden.

Nun bietet sich noch eine weitere Möglichkeit, die Geometrien im Suchbaum abzulegen ohne die o.g. Schwierigkeiten. Angenommen, man legt sowohl den x- als auch den y-Wert ab, so könnte man bereits im Baum selbst entsprechende „Ballungsräume“ herausfiltern. Der einfachste Ansatz wäre hier die Werte geeignet zu konkatenieren, beispielsweise mittels Multiplikation des x-Wertes mit einem entsprechend großen Faktor und anschließender Addition des y-Wertes. Zu sehen ist ein entsprechender Baum in Abbildung 3.3. Wie leicht zu erkennen ist, gibt es unter Beibehaltung der Einfügereihenfolge einige Unterschiede im linken Zweig des Baum.

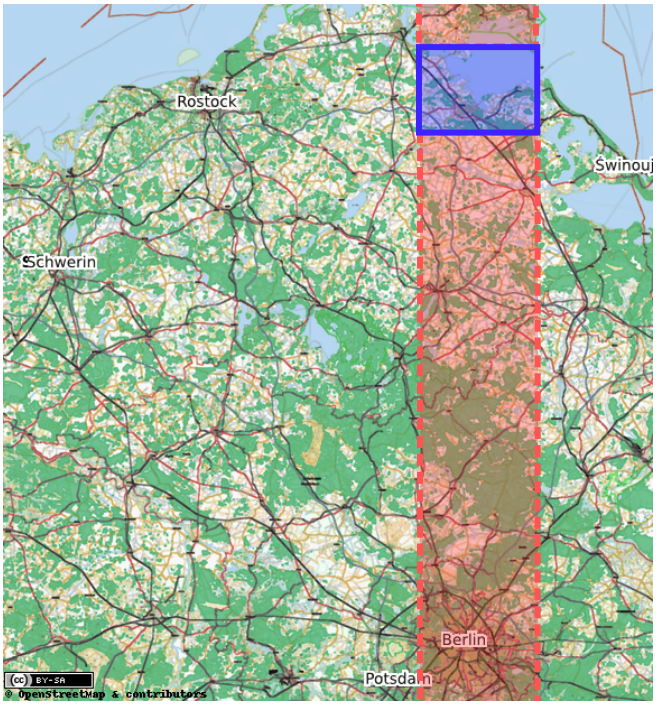

Abbildung 3.2: Darstellung der Probleme beim Indizieren von Geodaten mittels binärem Suchbaum

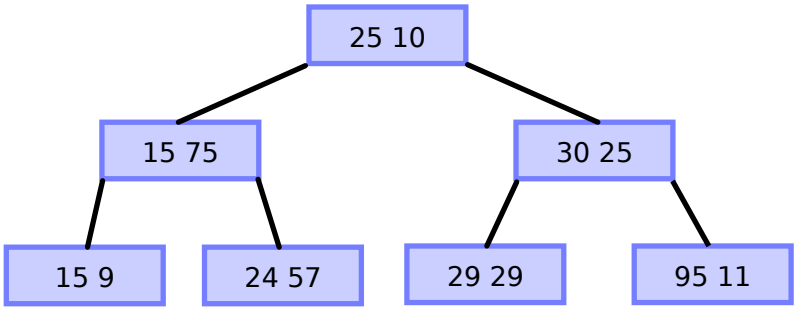

Abbildung 3.3: Binärer Suchbaum für Punktgeometrien. Kriterium ist eine Kombination von x- und y-Wert.

Aber auch diese Lösung ist nur in wenigen Anwendungsfällen praktikabel. Bereichssuchen profitieren letztlich nur sehr eingeschränkt, da zwischen zwei x-Werten letztlich alle y-Werte (mehrfach) liegen. Gewünscht wäre jedoch eine Form der Organisation, bei der x- und y-Wert „gleichberechtigt“ berücksichtigt werden können. Da es jedoch nicht möglich ist, eine entsprechende Ordnung auf zwei sich nicht gegenseitig bedingenden Attributen zu definieren, scheint es offenbar keine wirklich praktikable Lösung für die Verwendung des binären Suchbaums zu geben.

Doch letztlich trügt hier der Schein. Wenn es nicht (sinnvoll) möglich ist, beide Werte in einem Knoten als Schlüssel zu kombinieren, so ist es gleichwohl möglich, sie in unterschiedlichen Knoten jeweils getrennt als Schlüssel zu verwenden. Nun muss lediglich noch festgelegt werden, ob ein Knoten einen x-Wert oder einen y-Wert enthält. Das einfachste sich hier bietende Verfahren ist abhängig von der jeweiligen Tiefe im Baum einen der beiden Werte auszuwählen, beginnend mit dem x-Wert. Folglich differenziert die Wurzel nach x, die erste Ebene nach y, die zweite wieder nach x usw. Der sich hieraus ergebende Baum ermöglicht so eine relativ gleichmäßige Filterung nach x- und y-Wert. Es treten folglich nicht mehr die o.g. Problemfälle auf. In Abbildung 3.4 ist der entsprechende Beispielbaum zu sehen. Knoten mit x-Wert sind blau, solche mit y-Wert rot markiert. Der jeweils nicht relevante Wert steht in Klammern (die Reihenfolge ist immer x, dann y).

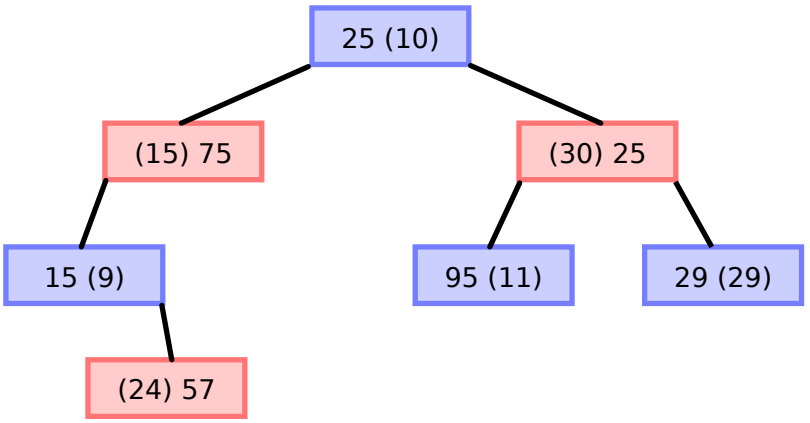

ABBILDUNG 3.4: kd-tree für Punktgeometrien

Diese Form des binären Suchbaums besitzt einen speziellen Namen, nämlich kd-Baum bzw. kd-tree. k und d stehen hierbei für die Mehrdimensionalität. k war der historische Bezeichner, im Sinne von k-dimensional. d wurde anschließend hinzugefügt, da dieser Bezeichner üblicherweise zur Angabe der Dimensionsanzahl verwendet wird[3].

3.2 Einfügen

Aus der Einführung ist bereits grundsätzlich erkennbar, wie Features in einen kd-tree eingefügt werden. Das Verfahren ist dabei sehr ähnlich dem des binären Suchbaums. Es sei angenommen, dass eine Punktgeometrie `P` eines Features in den Baum eingefügt werden soll. Ist dieser Leer, dann ist eine neue Wurzel anzulegen, den x-Wert des Punkts

[3] *Hanan Samet*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling), San Francisco, CA, USA 2005, S. 49.

als Schlüssel einzutragen und den Knoten mit dem zugehörigen Feature zu verknüpfen. Sollte der Baum jedoch nicht leer sein, muss dieser nach einer geeigneten Einfügestelle durchsucht werden. Dabei wird, ausgehend von der Wurzel, für jeden Knoten überprüft, ob der Schlüssel kleiner oder größer/gleich dem Wert der Punktgeometrie in der entsprechenden Dimension ist. Abhängig davon wird dann der linke oder rechte Teilbaum durchlaufen. Der Suchprozess beendet, wenn es keinen entsprechenden Teilbaum mehr gibt, womit gleichzeitig die Einfügestelle gefunden wurde. An dieser wird nun der entsprechende Knoten erzeugt und gefüllt.

Anzumerken ist, dass die für einen Knoten relevante Dimension für gewöhnlich nicht in Knoten selbst gespeichert wird. Der Suchprozess muss sich folglich die jeweils aktuelle Tiefe merken. Mittels `aktuelle_tiefe MOD d` wird dann die jeweils relevante Dimension bestimmt. Denkbar wäre auch eine Speicherung der relevanten Dimension im Knoten selbst möglich. Dies ist jedoch unter dem Gesichtspunkt der Laufzeit und des Speicherplatzverbrauchs nicht als sinnvoll zu erachten.

Einen Sonderfall stellt das Einfügen größerer Datenmengen dar. Dies erfolgt normalerweise immer dann, wenn für statische Geodaten ein Index erstellt werden soll. Hier ist es nicht sinnvoll diese sequenziell einzufügen. Zum einen, da die zu erwartende Laufzeit wegen der jeweils zu suchenden Einfügeposition relativ hoch ist. Zum anderen aber auch, da der resultierende Baum nicht unbedingt optimal sein muss. Daher wäre es sinnvoll, wenn die Daten als ganzes analysiert und eingefügt werden könnten.

Um dieses Ziel zu erreichen muss primär darauf geachtet werden, dass der jeweilige linke und rechte Teilbaum eines Knotens annähernd gleich groß ist. Hierfür entscheidend ist die Wahl eines geeigneten Schlüssels. Um diesen zu ermitteln bietet sich die Bestimmung des Medians auf der entsprechend der relevanten Dimension sortierten Datenmenge an. Anschließend wird dann noch das erste Feature gesucht, dessen Wert größer oder gleich dem Median ist. Alle links von diesem werden zum linken, alle rechts davon zum rechten Teilbaum. Das Feature selbst bildet den Knoten. Anschließend wird der Prozess rekursiv auf die linke und rechte Datenmenge angewendet.

Da jedoch auch der rein dynamische Ansatz in der Praxis seine Existenzberechtigung hat, wurden im Rahmen der Testumgebung beide Vorgehensweisen implementiert. Hieraus resultiert folglich ein statischer kd-tree und ein dynamischer kd-tree. Beide werden getrennt von einander im Kapitel 6 ausgewiesen.

3.3 Löschen

Anders als beim binären Suchbaum gestaltet sich das Löschen von Elementen aus einem kd-tree relativ aufwändig. Ursache hierfür ist die Iteration über die Dimensionen. Angenommen die Wurzel des in Abbildung 3.4 gezeigten kd-trees soll gelöscht werden. Folglich ist ein anderer, im Baum vorhandener Knoten, als neue Wurzel zu bestimmen. In einem binären Suchbaum wäre hier der Knoten 95/11 ein guter Kandidat. Im kd-tree sieht man allerdings sofort, dass er dies nicht ist. Durch den zyklischen Wechsel der relevanten Dimension kann nicht so einfach entschieden werden, welcher Knoten als geeignet angesehen werden kann, da auch jeweils die andere(n) Dimension(en) zu berücksichtigen sind. Dies gilt sowohl dahingehend, dass der „nächste (unmittelbare) Nachbar“ nicht der jeweils kleinste/größte Wert des jeweiligen Teilbaums sein muss (im Gegensatz im binären Suchbaum), als auch in der Weise, dass der resultierende Baum hinsichtlich aller Dimensionen konsistent zu sein hat.

Doch wie ist nun ein entsprechender Ersatz für den zu löschenden Knoten zu finden? Gesucht wird letztlich einer, dessen Wert in der relevanten Dimension dem aktuellen möglichst nahe kommt, also entweder der größte im linkten Teilbaum oder der kleinste im rechten Teilbaum ist. Nur leider genügt auch dies nicht vollständig. Denn gemäß der Bedingung für den linken und rechten Teilbaum darf der linke nur Werte **kleiner** als der Schlüssel enthalten. Wird nun der größte Wert als dem linkten Teilbaum gewählt ist jedoch nicht ausgeschlossen, dass dieser mehrfach in eben diesem vorkommt. Folglich würde es im linken Teilbaum der neuen Wurzel möglicherweise einen oder mehrere Werte geben, die **gleich** dem Schlüssel der Wurzel sind. Daher bleibt letztlich nur die Wahl des kleinsten Wertes in der jeweiligen Dimension aus dem rechten Teilbaum.

Nachdem dieser Knoten gefunden wurde, muss er jedoch zuerst selbst gelöscht werden, was wiederum den selben Prozess auslöst (Rekursion). Hierdurch wird aber gleichzeitig sichergestellt, dass auch die jeweils andere Dimension weiterhin konsistent ist, was bei einem reinen Verschieben des Knotens inklusive seiner Teilbäume nicht gegeben wäre. Wie leicht zu sehen ist terminiert der rekursive Löschvorgang, wenn ein Blatt als Ersatzknoten bestimmt wurde.

Nun mag der geneigte Leser jedoch einwenden, dass der bisher geschilderte Algorithmus zwar in der Regel gut funktioniert, es jedoch einen nicht berücksichtigten Sonderfall gibt. Was passiert, wenn ein Knoten gelöscht werden soll, der keinen rechten, aber sehr wohl einen linken Teilbaum besitzt? Wie bereits geschildert, ist es nicht möglich den größten Wert des linken Teilbaums zu wählen. Um genau zu sein ist es überhaupt nicht möglich einen Wert aus dem linken Teilbaum zu wählen, so lange es anschließen noch einen linken Teilbaum gibt. Doch gerade in diesem Dilemma liegt die Lösung. Was, wenn man einen

solchen Wert wählt, dass es anschließend so zu sagen keinen linken Teilbaum mehr geben dürfte, da alle Werte größer oder gleich diesem Wert sind. Formaler ausgedrückt wird also der kleinste Wert des linken Teilbaums gesucht. Beim Einfügen dieses Knotens (nachdem er an seiner ursprünglichen Position gelöscht wurde) wird dann aber der linke Teilbaum als rechter eingefügt, wodurch die Konsistenz des Gesamtbaums wieder hergestellt ist.

In Abbildung 3.5 wird der Vorgang noch einmal an einem Beispiel gezeigt. Ausgangspunkt ist, dass die Wurzel gelöscht werden soll (Schritt 1). Als Ersatz wurde der Knoten 15/9 ausgewählt (Knoten 15/75 wäre ebenfalls denkbar gewesen). Im zweiten Schritt wird nun der Knoten 15/9 gelöscht, dessen einzig möglicher Ersatz 24/57 ist. Nun muss im dritten Schritt auch dieser rekursiv gelöscht und anschließend neu eingefügt werden. Nun ist der Löschvorgang des ursprünglichen Ersatzknotens 15/9 fertig und im vierten Schritt wird dieser nun mit dem bisherigen linken Teilbaum verknüpft - jetzt allerdings als rechter Teilbaum.

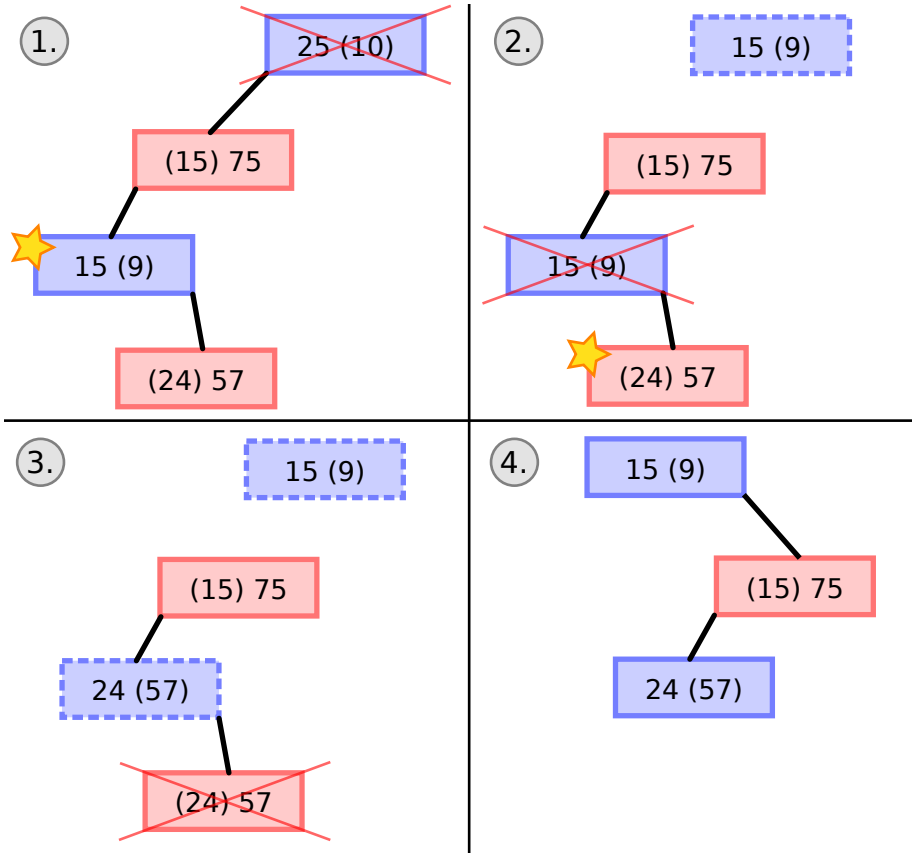

Abbildung 3.5: Löschen eines Knotens im kd-tree ohne rechtem Teilbaum

3.4 Suchen

Im Gegensatz zum Löschen gestaltet sich die Suche mittels Position oder Bounding Box wiederum relativ einfach. Ausgehend vom Wurzelknoten wird jeweils geprüft, ob der linke und/oder der rechte Teilbaum betroffen sind. Im Falle der Suche mittels Position muss logischerweise nur einer von beiden weiterverfolgt werden. Im Falle von Bounding Boxes können hingegen beide Seiten Relevanz besitzen, da hier jeweils eine Bereichssuche in der jeweiligen Dimension durchgeführt wird.

Zu beachten ist, dass die hier vorgestellte Variante des kd-trees Duplikate zulässt. Daher kann bei der Suche mittels Position nicht abgebrochen werden, wenn das erste Feature gefunden wurde. Der Abbruch der Suche erfolgt vielmehr wenn ein Blatt erreicht bzw. im Falle von Bounding Boxes, wenn in allen relevanten Teilbäumen jeweils die Blätter erreicht wurden.

Eine umfangreichere Behandlung verdient hingegen die Suche nach den nächsten Nachbarn. Hier ist jedoch nicht der primäre Grund der, dass ein besonders schwieriger oder trickreicher Algorithmus anzuwenden ist. Vielmehr soll hier nur initial der für alle realisierten Bäume angepasste Algorithmus in seiner grundlegenden Arbeitsweise vorgestellt werden. Konkret handelt es sich dabei um den inkrementell arbeitenden *Best-First Nearest Neighbor* Algorithmus[4][5].

Der Algorithmus ist in der Lage eine vorher nicht bekannte Anzahl an nächsten Nachbarn in sortierter Form zu ermitteln, beginnend mit dem Nächsten (*best-first*). Zwar wird in der im Rahmen der Testumgebung keine inkrementelle Ermittlung benötigt, da die gewünschte Anzahl vorab bekannt ist. In der Praxis kann es aber durchaus häufiger vorkommen, dass neben der Anzahl noch weitere Einschränkungen gefordert werden (z.B. die 5 nächstgelegenen Städte mit mehr als 100.000 Einwohnern).

Kernelement ist eine *Priority Queue*, also eine Schlange mit Priorisierung (Sortierung) der enthaltenen Elemente. Kriterium ist die Distanz[6] des jeweiligen Elements vom Ausgangspunkt der Suche wobei kleinere Distanzen bevorzugt werden. Zu Anfang enthält die *Queue* logischerweise nur die Wurzel. In einer Schleife wird nun jeweils ein Element aus der Schlange entnommen. Besitzt es keine Kinder handelt es sich um den nächsten

[4] *Samet*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling) (wie Anm. 3), S. 490ff.

[5] *Manolopoulos* u. a.: R-Trees: Theory and Applications (Advanced Information and Knowledge Processing) (wie Anm. 11), S. 59.

[6] Hier ist das Koordinatensystem zu berücksichtigen, in dem die Geometrien vorliegen. So kann es vorkommen, dass die Koordinaten als Längen-/Breitengrade vorliegen. Dementsprechend ist in einem solchen Fall auch sphärische Geometrie zur Distanzbestimmung zu bemühen. Alternativ sind alle Koordinaten vorab in ein geeignetes metrisches Koordinatensystem zu transformieren.

gefundenen Nachbarn. Besitzt dieses jedoch Kinder so sind diese in die Schlange einzufügen. In diesem Fall kann nun aber nicht das entnommene Element als nächster Nachbar verwendet werden, da sich die Schlange geändert hat. Folglich muss es nun kinderlos wieder eingefügt werden. Der gesamte Algorithmus wird in Abbildung 3.6 noch einmal in Pseudocode-Form dargestellt.

```
1  procedure IncrementalNearestNeighbor(Point position)
2      PriorityQueue queue
3      queue.enqueue(wurzel)
4
5      while not empty queue
6          Element e := queue.dequeue
7          if not e.hasChildren then
8              report e.feature
9
10         else
11             if e.hasLeftChild then
12                 queue.enqueue(e.leftChild)
13             end
14             if e.hasRightChild then
15                 queue.enqueue(e.rightChild)
16             end
17
18             Knoten eKinderlos := new Knoten(e.feature)
19             queue.enqueue(eKinderlos)
20         end
21     end
22 end
23
```

Abbildung 3.6: Best-First Nearest Neighbor Algorithmus für kd-trees

3.5 Bewertung

Der hier vorgestellte kd-tree ist zweifelsohne ein relativ leicht zu implementierender Index. Jedoch besitzt er auch eine Reihe von entscheidenden Nachteilen. Zu aller erst ist hier die reine Fokussierung auf einzelne Punktgeometrien zu nennen. Es ist nicht möglich LineStrings oder Polygone geschweige denn Mengen dieser Geometrien einzufügen.

Der zweite entscheidende Nachteil ist die Limitierung auf ein einzelnes Feature pro Knoten. Dies liegt darin begründet, dass der kd-tree die Objektmenge und nicht den Raum selbst teilt. Dies muss bei großen Datenmengen als ein relativ ungünstiges Verhalten betrachtet werden, da der resultierende Baum genauso viele Knoten besitzt wie Features in der Datenmenge enthalten sind. Dies sorgt zum einen dafür, dass bei der Suche sehr

viele Knoten geladen und durchlaufen werden müssen. Zum anderen steigt auch der Speicherverbrauch unnötig an, da für jeden Knoten zwei interne Referenzen verwaltet werden müssen (rechter und linker Teilbaum). Zumal zu erwarten ist, dass viele dieser Knoten selbst keine Kinder besitzen und somit die Referenzen leer sind[7]. Ebenfalls anzumerken ist in diesem Zusammenhang, dass die Reihenfolge, in der die Features in den Index eingefügt werden, entscheidend für dessen spätere Struktur ist. Auch dies begründet sich in der Unterteilung gemäß der Objekte und nicht des Raums.

Als drittes Problem ist dann noch zu erwähnen, dass ein kd-tree in der Regel nicht ausbalanciert ist. Kombiniert mit der Abhängigkeit von der Einfügereihenfolge können folglich entsprechend ungünstige Bäume entstehen. Im schlechtesten Fall degeneriert ein solcher Baum sogar zu einer linearen Liste, was alle Operationen entsprechend teuer macht.

Dies alles wirkt sich auch auf die Kosten für einen Löschvorgang aus, der wie bereits geschildert, relativ aufwändig ist. Der günstigste Fall ist, dass der zu löschende Knoten ein Blatt ist. Gleichzeitig ist dieser Fall allerdings auch der wahrscheinlichste, da wie bereits geschildert, der größte Teil der Knoten Blätter sind. Im ungünstigen Fall hängt die Laufzeit primär von der Suche nach einem geeigneten Ersatzknoten ab. Bei `N` Knoten im Teilbaum und `d=2` Dimensionen beträgt diese $O(N^{1-\frac{1}{d}})$[8].

Gleichwohl soll nicht verschwiegen werden, dass sich viele diese Kritiken auf den Worst-Case beziehen. Im Durchschnittsfall hingegen weisen die Such- und Einfügeoperationen ein deutlich günstigeres Verhalten auf. So zeigt das Einfügen und die Suche nach einer Position bei `N` Knoten lediglich Kosten von $O(log_2 N)$ auf[9].

3.6 Four-dimensional kd-tree

Wie geschildert ist einer der Hauptnachteile des kd-trees der, dass lediglich Punktgeometrien indiziert werden können. Nun sind LineStrings und Polygone im Kontext von Geodaten nicht selten. Daher wäre es wünschenswert den kd-tree entsprechend zu adaptieren, wobei es genügt, die Geometrien in Form von MBRs, also Rechtecken, im Index zu verwalten.

Hierfür existieren im wesentlichen zwei Ansätze. Der erste besteht darin, den kd-tree so zu modifizieren, dass dieser den Raum unterteilt (trie). Anschließend kann dann ein

[7] *Samet*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling) (wie Anm. 3), S. 49.
[8] Ebd., S. 53.
[9] Ebd., S. 51.

Rechteck in einem oder mehreren Knoten gespeichert werden. Abschnitt 3.7 beleuchtet diese Variante näher. Der andere Ansatz verzichtet darauf, Rechtecke explizit als Geometrie zu unterstützen. Dies mag im ersten Moment verwirrend wirken, doch gibt es noch eine weitere Möglichkeit Rechtecke zu indizeren.

Ein Rechteck kann entweder durch zwei Punkte oder durch einen Punkt sowie einer Breiten- und Höhenangabe konstruiert werden. Es sind also vier unterschiedliche Werte notwendig (x1, y1, x2, y2 oder x, y, w, h). Der Trick besteht nun darin, diese vier Werte als Koordinatenangaben eines vierdimensionalen Punkts zu betrachten. Ein solcher spezieller Punkt wird als als repräsentativer Punkt bezeichnet[10].

Ausgehend hiervon ist leicht zu erkennen, dass eine entsprechende Verwendung im kd-tree mit relativ geringem Aufwand möglich ist. Die im Rahmen der Testumgebung realisierte Implementierung entspricht auch exakt diesen Vorgehen. Dies ist deshalb von Bedeutung, da in der Literatur bei der Verwendung von repräsentativen Punkten auch gewisse Abweichungen vor allem hinsichtlich der Indizierung größerer statischer Datenmengen vorgeschlagen werden. Der daraus resultierende Baum wird als *Balanced four-dimensional kd-tree* bezeichnet[11]. Im Rahmen dieser Thesis wurde jedoch mit Bedacht lediglich die Verwendung repräsentativer Punkte realisiert, da die restlichen Änderungen keinen direkten Zusammenhang mit eben diesen haben. Es handelt sich vielmehr dabei um allgemeine Modifikationen zwecks Optimierung der Baustruktur. Von diesen existiert jedoch eine Vielzahl weiterer möglicher.

Der hier vorgestellte vierdimensionale kd-tree unterscheidet sich gegenüber dem klassischen kd-tree in zwei Dingen. Erstens kommen nun vier Dimensionen zu tragen, wodurch sich die einzelnen Dimensionen nur noch alle vier Ebenen im Baum wiederholen. Hierfür sind aber keine Anpassungen an den Algorithmen, sondern allenfalls in der Implementierung durchzuführen. Zweitens müssen die Suchoperationen entsprechend modifiziert werden, wodurch es in diesem Bereich folglich dennoch zu gewissen Änderungen an den Algorithmen kommt. Hier sind im Wesentlichen nur die Bounding Box-Abfrage sowie die Nearest Neighbor-Ermittlung betroffen. In beiden Fällen ist die Eingrenzung des Suchraums über die Breiten-/Höhenangaben in der dritten und vierten Dimension zu beachten. Die dabei vorzunehmenden Änderungen können mit relativ wenig Aufwand durchgeführt werden.

[10] *Samet*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling) (wie Anm. 3), S. 453ff.
[11] Ebd., S. 460.

3.6.1 Bewertung

Der Weg über repräsentative Punkte komplexere Geometrien zu unterstützen ist ein sehr einfach umzusetzender. Daher harmoniert er relativ gut mit einer der stärken des kd-trees. Er bietet jedoch auch einige grundsätzliche Nachteile. Durch die zusätzlichen Dimensionen werden bei Suchoperationen entsprechend weniger Teilbäume verworfen. Ursächlich hängt dies damit zusammen, dass die Eingrenzung der Position von Geometrien besser als Kriterium bei der Suche verwendet werden kann als das Wissen um die Größe eben dieser. Da nun zwei Dimensionen allein solche Informationen enthalten ist dies eher ungünstig. Gleichwohl gibt es auch Suchoperationen bei denen dies von Vorteil sein dürfte. Hier sind die Relationsoperationen *equals* und *covers* zu nennen, bei denen über die Größe relativ gut Teilbäume verworfen werden können. Bei den klassischen Operationen wie der *Window Query* und insbesondere auch der *Nearest Neighbor Query* ist dies jedoch ein klarer Nachteil.

3.7 Kdb-tree

Die beiden bisher diskutierten Varianten des kd-trees sind in vielerlei Hinsicht gleich oder zumindest sehr ähnlich. Nun soll dem als dritter Index ein anderer Ansatz gegenüber gestellt werden. Vor allem zwei Dinge sind dabei besonders zu betrachten. Die beiden bisherigen Varianten basieren auf dem binären Suchbaum und unterteilen nicht den Suchraum an sich sondern die Objektmenge. Der geneigte Leser hat sich vielleicht schon bei der Vorstellung des kd-trees gefragt, ob nicht statt des binären Suchbaums ein B-Tree besser geeignet wäre. Genau dieser Ansatz soll hier nun weiter verfolgt werden.

Der B-Tree (und seine Varianten) bietet gegenüber anderen Indices eine Reihe von Vorteilen, die ihn gerade im Bereich der Datenbankmanagementsysteme sehr populär gemacht hat. Primär zwei dieser Vorteile seien hier genannt, da sie in den weiteren Ausführungen von Bedeutung sind. Erstens die vollständige Ausbalanciertheit, die insbesondere der Degenerierung des kd-trees entgegenwirkt, und zweitens die Unterstützung von Buckets, also der Speicherung von mehr als einem Feature/zwei Kindern pro Knoten. Genau wie beim binären Suchbaum lässt er sich aber nicht direkt für Geodaten verwenden. Es ist aber möglich die selben Techniken anzuwenden, die auch zur Adaption von diesem verwendet wurden. Der aus diesen Anpassungen sowie der Umstellung auf die Unterteilung des Suchraums resultierende Baum wird als kdb-tree bezeichnet[12].

[12] *John T. Robinson*: The K-D-B-tree: a search structure for large multidimensional dynamic indexes, in: Proceedings of the 1981 ACM SIGMOD international conference on Management of data (SIGMOD '81), Ann Arbor, Michigan 1981, S. 10–18.

Um nun die erwähnte Unterteilung des Suchraums (Trie) statt der Objektmenge durchzuführen, sind jedoch einige Veränderungen notwendig. Es beginnt damit, dass statt eines klassischen B-Trees ein B+-Tree verwendet wird. Dies ermöglicht, die inneren Knoten mit „beliebigen“ Schlüsseln auszustatten, da die eigentlichen Features lediglich auf der Ebene der Blätter referenziert werden. Folglich gibt es also einen konzeptionellen Unterschied zwischen den Blättern, welche eine Menge Punktgeometrien enthalten und über diesen ein Rechteck aufspannen sowie den inneren Knoten, die selbst eine Menge von Rechtecken enthalten und darüber ein eben solches bilden. Abbildung 3.7 zeigt diesen Sachverhalt sowohl hinsichtlich der Flächen (links) als auch bezüglich des Baums (rechts). Die gewählten Farben korrespondieren zwischen beiden Darstellungen.

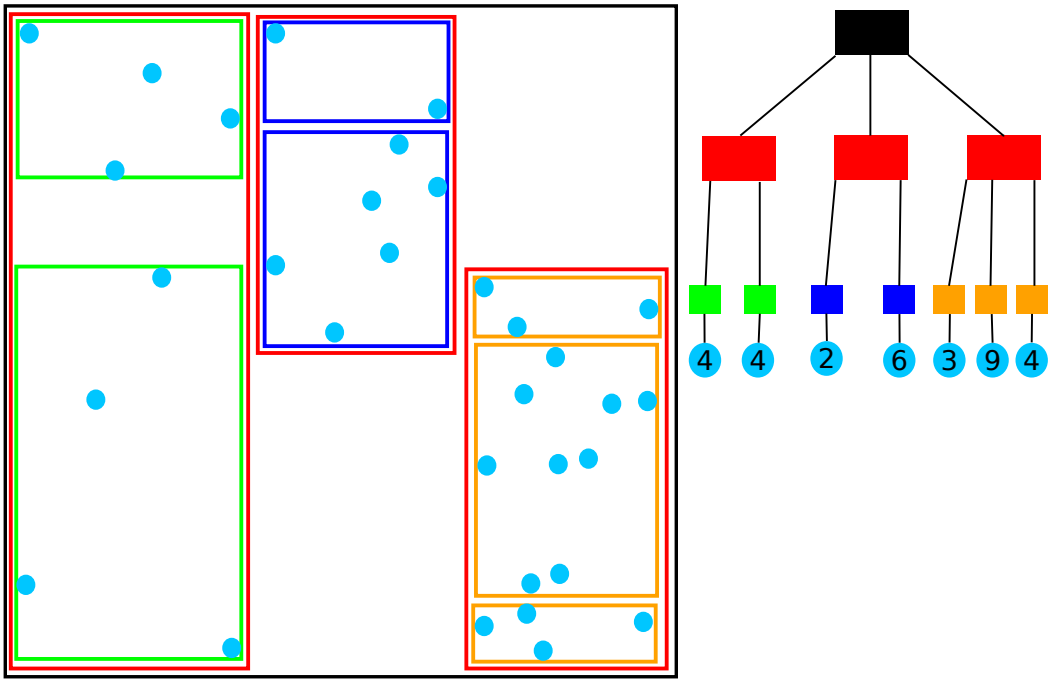

Abbildung 3.7: Beispiel der grundsätzlichen Struktur eines kdb-trees hinsichtlich der Punktgeometrien und der dadurch aufgespannten Flächen

Diese Klassifizierung hat weitgehende Folgen. Betrachtet man nur die inneren Knoten (Rechtecke), so lässt sich eine gewisse Ähnlichkeit zum R-Tree feststellen (siehe Kapitel 5). Gleichwohl weichen die verwendeten Algorithmen von einander ab. Eine recht treffende alternative Bezeichnung für den kdb-tree liefert in diesem Zusammenhang Samet[13]. Dort wird der Index als *generalized k-d tree bounding-box cell-tree pyramid* bezeichnet. Diese zuerst relativ sperrige Bezeichnung fasst aber gerade die Klassifizierung in Blätter

[13] *Samet*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling) (wie Anm. 3), S. 306.

(Punkte) und innere Knoten (Rechtecke) elegant zusammen. Dabei bezieht sich der Term *bounding-box cell-tree pyramid* auf die Struktur der inneren Knoten und ihrer Hierarchie.

Leicht zu erkennen ist, dass durch diese Struktur des Baums alle Manipulationsoperationen signifikante Unterschiede zum kd-tree aber auch zum B+-Tree aufweisen. Hier sei insbesondere die Teilung von Knoten sowie das Löschen von Elementen und der damit verbundenen Reorganisation zu nennen. Daher soll nun zuerst auf diese Operationen eingegangen werden, bevor anschließend die Suche folgt. Anschließend wird eine Adaption für Nicht-Punktgeometrien vorgestellt.

3.7.1 Einfügen

Das grundsätzliche Vorgehen beim Einfügen in einen kdb-tree ist ausgehend von B-Tree und dem kd-tree leicht zu erkennen. Im einfachsten Fall ist der Baum leer. Dann genügt es ein neues Blatt zu erzeugen und das Feature dort einzufügen. Wenn nun aber schon ein Wurzelknoten existiert, muss eine entsprechende Einfügeposition ermittelt werden. Dazu der Baum analog dem kd-tree durchlaufen und immer genau der Teilbaum gewählt in dessen Bereich die Geometrie des einzufügenden Features liegt. Der Vorgang terminiert, wenn ein Blatt erreicht wurde. Im günstigen Fall kann nun das Feature dort eingetragen werden. Im ungünstigen Fall ist hingegen ein Split des Blatts notwendig, da dessen Kapazität sonst überschritten würde.

Im Falle eines Blattes kann ein Split noch mit relativ wenig Aufwand durchgeführt werden. Zuerst ist die relevante Dimension zu bestimmen. Anschließend ist für diese Dimension eine geeignete Position zu ermitteln, so das beide neuen Knoten möglichst gleich viele Features enthalten. Doch sollte eine gewisse Vorsicht geboten sein, da es vorkommen kann, dass in der gewählten Dimension kein Split möglich ist. Dies tritt immer dann ein, wenn alle Punktgeometrien in dieser Dimension den selben Wert haben. In einem solchen Fall muss folglich die jeweils andere Dimension zum Teilen verwendet werden[14]. An dieser Stelle ist noch einmal kurz an den kd-tree zu erinnern. In diesem wurde die Dimension nicht in den Knoten gespeichert, da aus der momentanen Tiefe im Baum eben diese abgeleitet werden konnte. Im kdb-tree ist dies aber wegen dem gerade geschilderten Sachverhalt nicht möglich. Folglich muss die relevante Dimension in den Knoten abgelegt werden. Der Vorteil hiervon ist wiederum, bei einem Split potentiell mit der günstigsten Dimension arbeiten zu können.

[14] Dies ist nur uneingeschränkt möglich, wenn keine gleichen Punktgeometrien im Baum erlaubt werden. Sind diese hingegen möglich, so kann es passieren, dass kein Split durchgeführt werden kann. In einem solchen Fall muss die Kapazität des Knotens auf geeignetem Wege vergrößert werden. Da dies im Rahmen der Testumgebung der Fall ist, wurden entsprechend verkettete Swappages realisiert.

Nun kann ein Split eines Blattes anschließend auch zum Split eines inneren Knotens führen, wenn nun seinerseits ein solcher zu voll geworden ist. Hier ist die Teilung ungleich komplizierter. Der Grund ist leicht zu sehen, wenn man sich die Konsequenzen der freien Wahl der jeweils relevanten Dimension vor Augen führt. So ist es ohne Probleme möglich, dass die Kinder eines inneren Knotens nicht einheitlich entsprechend einer Dimension geteilt wurden. Abbildung 3.8 zeigt einen solchen Knoten (schwarz) sowie seine sechs Kinder (rot, A-F). Zu sehen ist, dass kein Split möglich ist, ohne dabei einen bestehenden Knoten zu teilen. Also bleibt folglich keine andere Möglichkeit als einen oder mehrere Knoten zu zerschneiden. Eine mögliche Variante ist in der Abbildung in Blau angedeutet. Hier wird das Kind F in zwei Hälften gespalten.

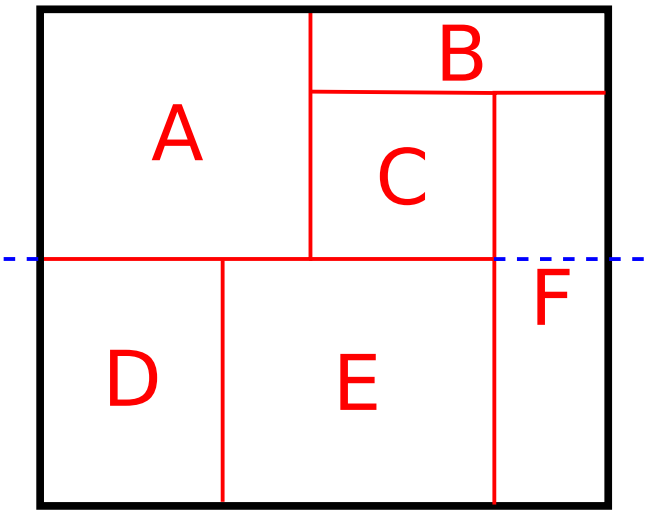

Abbildung 3.8: Beispiel für das Split-Problem bei inneren Knoten im kdb-tree

Bei der Teilung eines Knotens in Folge eines Splits muss dieser Vorgang ggf. auf die Kinder dieses Knotens rekursiv angewendet werden, bis die Blätter erreicht sind. Hierdurch kann es vorkommen, dass die resultierenden Knoten nicht optimal gefüllt sind. Anzumerken bleibt noch, dass bei einem Split der Wurzel analog zum B-Tree ein neuer Wurzelknoten erstellt werden muss. Die aus dem Split resultierenden Knoten werden dann zu Kindern der neuen Wurzel.

3.7.2 Löschen

Prinzipiell gestaltet sich das Löschen eines Features aus dem kdb-tree recht einfach. Es genügt dieses im Baum zu suchen und es anschließend zu entfernen. Wenn der Knoten, in dem es referenziert war, in Folge dessen leer wird, ist dieser ebenfalls zu entfernen. Der Prozess ist entsprechend rekursiv fortzusetzen.

Ähnlich der Zwangsteilung von Knoten in Folge eines Splits beim Einfügen, können auf diese Weise allerdings sehr ungünstige Bäume entstehen, bei denen ein Großteil der Knoten nur zu einem geringen Prozentsatz gefüllt ist. Daher wird im Rahmen des kdb-trees vorgeschlagen[15], nach dem Löschvorgang eine Reorganisation der betroffenen Knoten durchzuführen. Hierzu ist eine zu wenig gefüllte Seite A mit einem anderen Kind B des selben Elternknotens P zu verschmelzen. Doch hier ist etwas Vorsicht geboten. Aus den selben Gründen, wegen denen beim Split unter Umständen eine Zwangsteilung durchgeführt werden muss, kann auch das Verschmelzen ein Problem darstellen. Zu bedenken ist nämlich, dass die resultierende „Geometrie“ auch wieder ein Rechteck sein muss. Aber selbst bei direkt benachbarten Knoten ist dies nicht automatisch zu gewährleistet. Aus diesem Grund müssen in einigen Fällen mehr als zwei Knoten verschmolzen werden um ein entsprechendes Rechteck zu erhalten. Im ungünstigsten Fall können dies alle Kinder von P sein

Der aus diesem Vorgang resultierende Knoten kann, wie leicht zu erkennen, zu voll sein. Daher ist es anschließend notwendig diesen wieder in entsprechend gefüllte Teilknoten zu zerlegen (Split). Falls diese nicht auf der Ebene der Blätter sondern einer Ebene von inneren Knoten passieren, so kann dies wiederum zur Teilung von Kindknoten führen. Die Folge hiervon kann dann allerdings ein schlechterer Füllgrad sein als vor der Reorganisiation. Die Reorganisation ist also kein ganz unproblematischer Prozess. Eine Möglichkeit diese Gefahr zu reduzieren ist, nach einem Split ebenfalls eine solche durchzuführen. Doch auch dieses Verfahren hat seine Tücken, da folglich eine Reorganisation einen Split auslösen kann, der wiederum eine Reorganisiation auslöst. In der Folge kann eine einzelne Reorganisation eines relativ hoch angesiedelten Knotens eine ganze Kaskade von Änderungen im gesamten Baum auslösen. Im Rahmen der Testumgebung wurde diese daher nur im Folge von Löschvorgängen realisiert.

3.7.3 Verschieben

Anders als beim kd-tree ist es im kdb-tree möglich, das Verschieben von Features nicht nur über eine Lösch- gefolgt von einer Einfügeoperation abzubilden. Da die Blätter, in denen die Features gespeichert werden, jeweils ein Rechteck bilden, kann es vorkommen, dass das Feature auch nach dem Verschieben sich weiterhin im selben Blatt befindet. Von daher ist es lohnenswert zuerst zu prüfen, ob lediglich eine solche kleine Änderung durchgeführt werden muss. Sollte es hingegen nicht möglich sein bleibt nur der Fallback auf das bekannte Verfahren.

[15] *Robinson*: The K-D-B-tree: a search structure for large multidimensional dynamic indexes (wie Anm. 12), S. 15.

3.7.4 Suche

Die Suchoperationen sind ähnlich denen des kd-trees. Es sind aber ein paar Besonderheiten zu beachten. So muss bei inneren Knoten nicht die relevante Dimension sondern die gesamte Bounding Box des Kinds überprüft werden. Für die Location und Window Query muss dazu jeweils geprüft werden, ob sich die Suchgeometrie mit dem jeweiligen Rechteck des Kindes schneidet. Einfacher wird es hingegen für die Nearest Neighbor Operation, da die Distanzberechnung direkt gegen dieses Rechteck erfolgen kann. Es ist also nicht mehr notwendig, aus dem Schlüsselelement und seiner Eltern ein entsprechendes Gebiet abzuleiten.

Ebenfalls zu berücksichtigen ist, dass alle Features nun in den Blättern abgelegt sind. Dies stellt allerdings nur eine minimale Anpassung dar und ist, ebenso wie die Berücksichtigung von mehr Kindern pro Knoten, entsprechend leicht zu implementieren.

3.7.5 Unterstützung von LineStrings und Polygonen

Wie der kd-tree unterstützt auch der kdb-tree von sich aus nur Punktgeometrien. Möglich wäre nun eine Adaption der repräsentativen Punkte, um entsprechend komplexere Geometrien indizieren zu können. An dieser Stelle soll jedoch eine andere Variante vorgestellt werden, welche relativ leicht aus den Gegebenheiten abgeleitet werden kann. Zudem kann diese Variante auch besser mit Punktgeometrien umgehen als der bereits vorgestellte four-dimensional kd-tree.

Da der kdb-tree ohnehin bereits an vielen Stellen mit Rechtecken arbeitet, liegt es auf der Hand, statt Punkten auch andere Geometrien als MBRs einzufügen. Die Blätter bilden dann kein Rechteck aus einer Menge von Punkten sondern aus einer Menge von MBRs dieser Geometrien. Doch ist dies nur ein Teil der notwendigen Adaptionen. Denn schwierig wird es dann, wenn sich die Geometrie eines einzufügenden Features über mehrere bestehende Blätter erstreckt. Dies kann ohne weiteres eintreten, da die durch die Blätter aufgespannten Rechtecke überlappungsfrei sein müssen (ein wesentlicher Unterschied zum r-tree). Aufwändiger ist auch die Adaption des Split-Algorithmus, da auch hier durch die Zwangsteilung von Knoten Situationen eintreten können, in denen sich Features über mehrere Blätter erstrecken.

Nun gibt es zwei mögliche Lösungen. Zum einen wäre es denkbar, Überlappungen von Knoten zuzulassen. Hierdurch würde sich der kdb-tree jedoch inhaltlich sehr weit vom kd-tree entfernen. Zum anderen ist es möglich, ein Feature in so vielen Knoten wie notwendig zu speichern. Dieser Ansatz wurde im Rahmen der Testumgebung realisiert.

Auf den ersten Blick scheint der Umfang der durchzuführenden Modifikationen gering zu sein. Die Einfügeoperation muss dahingehend angepasst werden, dass bei der Suche nach der Einfügeposition mehrere Blätter ermittelt werden können. Gleiches gilt für die Löschoperation. Es genügt nun nicht mehr nur den erstbesten Knoten zu finden. Auch die Suchoperationen sind auf ähnliche Weise anzupassen, nur dass hier darauf zu achten ist, dass ein Feature auch nur ein einziges Mal gegenüber dem Aufrufer „gefunden“ wird. Die Verschiebeoperation kann hier ignoriert werden, da die Testumgebung lediglich Punktgeometrien im Kontext der Bewegungssimulation unterstützt.

Ein Problem stellt der Split von Knoten dar. Im Falle von Punktgeometrien wurde bereits erwähnt, dass die Wahl der Dimension nicht beliebig erfolgen kann. Vielmehr kann es passieren, dass abhängig von der Verteilung der Geometrien eine Dimension ausscheidet. Im Fall von Nicht-Punktgeometrien verschärft sich dieses Problem weiter. Nun kann es regelmäßig passieren, dass ein Split überhaupt nicht möglich ist, da die MBRs aller Geometrien die Fläche des Knotens komplett ausfüllen. Demgegenüber sei an die Punktgeometrien erinnert, wo dies nur bei vielen Duplikaten auftreten kann. Es bleibt also nur die Möglichkeit den jeweiligen Knoten entsprechend zu vergrößern. Im Rahmen der Testumgebung wurde dazu die bereits für Punktgeometrie-Duplikate realisierte Unterstützung von Swap-Pages verwendet. Dazu besitzt jeder Knoten eine Referenz auf (mögliche) Swap-Page, die weitere Elemente enthalten kann. Ist diese ebenfalls voll, so kann eine solche ebenfalls wieder auf eine Swap-Page verweisen.

Zu beachten ist beim Split auch, dass die Suche nach einer passenden Position in einer Dimension nicht ganz trivial ist. Erstens ist zu berücksichtigen, dass die MBRs unter Umständen größer sind als der Knoten selbst. Zweitens sind die Überlappungen der MBRs untereinander zu berücksichtigen. Es muss folglich eine Position gefunden werden, die nicht innerhalb eines MBRs liegt. Gleichzeitig muss sie aber auch innerhalb des Knotens liegen, wobei auch die Kanten des Knotens keine erlaubte Position darstellen, da sonst ein Rechteck mit der Fläche 0 entstehen würde.

Problematisch ist ferner, dass ja beim sequenziellen Einfügen in mehrere Knoten (Einfügepositionen) auch entsprechend viele Splits durchzuführen sind. Dies kann ferner bedeuten, dass durch einen solchen Split sich die Menge der noch zu betrachtenden Einfügepositionen ändert. Daher ist beim Split zu prüfen, ob ein entsprechender Knoten sich in dieser Menge befindet. Nach Durchführung des Splits ist dieser dann durch die beiden neuen Knoten zu ersetzen. Ebenso schwierig ist auch die Verwaltung der Pfade zur Wurzel hin. Immerhin ist bei einem Split auch der jeweilige Elternknoten zu berücksichtigen, was wiederum allerdings ein Split auslösen kann. Um das Problem im Rahmen der Testumgebung zu vereinfachen wurde ein Cache aller für die konkrete Einfügeoperation bereits geladener Knoten eingeführt. Entsprechende Änderungen werden dort verbucht. Ist nun

ein Elternknoten zu ermitteln, so wird dieser Cache entsprechend durchsucht. Hierdurch kann das komplizierte mithalten der jeweiligen Pfade für jede Einfügeposition vermieden werden, welche letztlich auch nur die Knoten vorgehalten hätten, die sich auf diese Weise im Cache befinden. Es sei aber ausdrücklich darauf hingewiesen, dass der Cache nicht über mehrere Operationen hinweg arbeitet sondern immer für genau eine Operation instantiiert wird.

Aufwändig scheint auch die Adaption der Nearest Neighbor-Operation. An dieser Stelle lohnt sich allerdings vorab einen Blick auf andere Indexalgorithmen zu werfen. Insbesondere der r^+-tree ist hier als prominentes Beispiel zu nennen[16], da auch hier ein Feature mehrfach im Baum vorhanden ist. Ausgehend hiervon ist es nun leicht entsprechende Adaptionen des *best-first nearest neighbor*-Algorithmus zu finden. Die hier vorgestellte Variante stammt von Samet[17].

Zu lösen sind bei genauer Betrachtung zwei Probleme in der Nearest Neighbor-Operation. Erstens sollten all jene Duplikate erkannt werden, die bereits Teil der Schlange (priority queue) sind. Zweitens soll vermieden werden, solche Features in die Schlange einzufügen, die bereits an den Aufrufer gemeldet wurden. Zu sehen ist die entsprechende Realisierung in Abbildung 3.9. Dabei können die Zeilen 8 bis 10 dem ersten Problem und die Zeilen 16 bis 21 dem zweiten zugeordnet werden.

Ersteres wird dadurch gelöst, dass wenn ein Feature der Schlange entnommen worden ist (Zeile 7) anschließend auch alle Duplikate dieses Features entnommen werden (Zeile 8 bis 10) . Dies ist einfach möglich, da alle Duplikate logischerweise die selbe Distanz zum Ausgangspunkt haben und sich folglich am Anfang der Schlange befinden.

Letzteres Problem ist durch eine entsprechende Distanzprüfung beim Einfügen von Features zu vermeiden (Zeile 19). Dabei sind nur solche Features einzufügen, deren Distanz zum Ausgangspunkt größer oder gleich der ihres Blattknotens ist. Ist die Distanz nämlich kleiner, muss das Feature auch Teil eines anderen Blattes sein, dass näher zu der Position liegt und folglich bereits „gefunden“ wurde.

Ferner sind zwei Annahmen bezüglich der Sortierung in der Schlange zu machen. Als erstes wird vorausgesetzt, dass bei gleicher Distanz zum Ausgangspunkt Features entsprechend ihrer ID bzw. ihrer Nummer sortiert werden. Auf diese Weise wird sichergestellt, dass gleiche Features (Duplikate) immer unmittelbar hintereinander folgen und somit durch den Code den Zeilen 8 bis 10 erfasst werden können. Die zweite Annahme ist, dass bei gleicher Distanz Knoten gegenüber Features bevorzugt werden. Andernfalls könnte

[16] *Manolopoulos* u. a.: R-Trees: Theory and Applications (Advanced Information and Knowledge Processing) (wie Anm. 11), S. 15.
[17] *Samet*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling) (wie Anm. 3), S. 499.

```
1  procedure IncrementalNearestNeighborDuplicate(Point pos)
2     PriorityQueue queue
3     queue.enqueue(wurzel)
4
5     while not empty queue
6        Element e := queue.dequeue
7        if not e.hasChildren then
8           while e = queue.peek
9              e.dequeue
10          end
11
12          report e.feature
13
14       else
15          for child in e.children
16             if not child.hasChildren then
17                queue.enqueue(child)
18
19             else if child.distance(pos) >= e.distance(pos) then
20                queue.enqueue(child)
21             end
22          end
23       end
24    end
25 end
```

Abbildung 3.9: Best-First Nearest Neighbor Algorithmus mit Duplikatserkennung für kdb-trees

es passieren, dass ein Feature A gemeldet wird, obwohl es noch einen Knoten K mit der selben Distanz gibt, in dem es ebenfalls enthalten ist. Wird nun K verarbeitet, ist es an keiner Stelle des Algorithmus mehr möglich zu erkennen, dass A bereits verarbeitet wurde. Folglich würde dann das selbe Feature mehrfach gemeldet. Dem Verständnis halber sei noch einmal angemerkt, dass es unkritisch ist, wenn weiter entfernte Knoten ebenfalls A enthielten. Dies würde in der Zeile 19 erkannt werden.

3.7.6 Bewertung

Durch die Verwendung des B-Trees ist die Performance des kdb-trees entsprechend zuverlässiger. Insbesondere die Degenerierung des kd-trees kann hier konstruktionsbedingt nicht stattfinden. Auch die Unterstützung von mehreren Features/Kindern pro Knoten ist entsprechend positiv einzuschätzen, da auf diese Weise die Gesamtanzahl der Knoten sowie die Tiefe des Baums sinkt. Hierdurch müssen weniger Knoten von Datenspeicher gelesen werden, was eine entsprechend günstigere Laufzeit der Suchoperationen erwartet lässt.

Deutlich aufwändiger ist hingegen das Einfügen sowie Löschen von Knoten, zum einen durch den Split und zum anderen durch die Reorganisation. Hier muss allerdings beachtet werden, dass gerade diese Operationen zu der insgesamt besseren Organisation des Baums beitragen. Leider gibt es in der Literatur keine grundsätzlichen Aussagen zu der zu erwartenden Laufzeit. Experimentelle Ergebnisse legen aber nahe, dass diese als eher günstig einzustufen ist[18].

Kritischer ist die Unterstützung für Nicht-Punktgeometrien zu beurteilen. Zwar ist es prinzipiell möglich, wie hier geschehen, auch den kdb-tree entsprechend zu modifizieren. Aber insgesamt bleiben die Auswirkungen auf die Laufzeit schwer einzuschätzen. Problematisch kann sich in diesem Zusammenhang die Duplizierung von Features herausstellen. Dies verkompliziert die Löschoperation und insbesondere auch den Split. Die Auswirkungen auf die Reorganisation sind schwer einzuschätzen. Da aber auch hier entsprechende Splits durchgeführt werden müssen ist eine eher kritische Tendenz anzunehmen. Insgesamt bleibt abzuwarten, wie die experimentellen Ergebnisse den kdb-tree im Vergleich zu anderen Indices erscheinen lassen.

[18] *Robinson*: The K-D-B-tree: a search structure for large multidimensional dynamic indexes (wie Anm. 12), S. 16ff.

Kapitel 4

quadtrees

4.1 Einführung

Im Rahmen des Kapitels 3 wurde bereits darauf eingegangen, wie mehrdimensionale Daten (Punktgeometrien) in einem binären Suchbaum abgelegt werden können. Die dort präsentierte Lösung bestand darin, in einem Knoten nur eine Dimension zu betrachten. Dies war eine logische Folge daraus, dass auf mehrdimensionalen Geometrien „kleiner/-gleich“ bzw „größer“ nicht praktikabel definiert werden kann und somit ein Feature nicht einfach einem linken oder rechten Teilbaum zugeordnet werden kann.

In diesem Kapitel soll nun eine andere Lösung für genau dieses Problem präsentiert werden. Statt die Anzahl der betrachteten Dimensionen entsprechend zu reduzieren, kann auch eine andere Herangehensweise gewählt werden. Das Problem ist dabei, dass ein Feature (Geometrie) einem Teilbaum zugeordnet werden muss. Dabei ist das Referenzobjekt (Schlüssel des Knotens) ebenfalls eine solche Geometrie. Betrachten wir also die verschiedenen Resultate, die bei einem „Vergleich“ zwischen beiden auftreten können. Es können jeweils die x- und y-Werte kleiner, gleich oder größer sein. Betrachtet man nur einen von beiden Werten, ist also eine entsprechende Zuordnung möglich, womit wir wieder beim kd-tree wären. Es gibt aber noch einen weiteren Weg. Genau genommen ist nämlich eine entsprechende Zuordnung auch bei Verwendung beider Werte möglich! Voraussetzung hierfür ist allerdings, dass die Anzahl der Teilbäume erhöht wird.

Letztlich muss jedem Teilbaum eine bestimmte Kombination bzgl. der Ordnung der Einzelwerte zueinander zugeordnet werden. Dabei bietet es sich an, wieder nur zwei Varianten, nämlich kleiner/gleich und größer zuzulassen[1]. Daraus resultieren dann vier verschiedene Kombinationen, welchen jeweils der Einfachheit halber ein Name zugeordnet wird. Für diese hat sich die Verwendung der Himmelsrichtungen bzw. ihrer englischen Abkürzungen eingebürgert. Allgemein werden die „viertel“ eines auf diese Weise unterteilten Knotens auch als Quadranten bezeichnet, woraus sich dann die Bezeichnung für den Index selbst ableitet: quadtree. Abbildung 4.1 zeigt die genaue Zuordnung sowie die Namen.

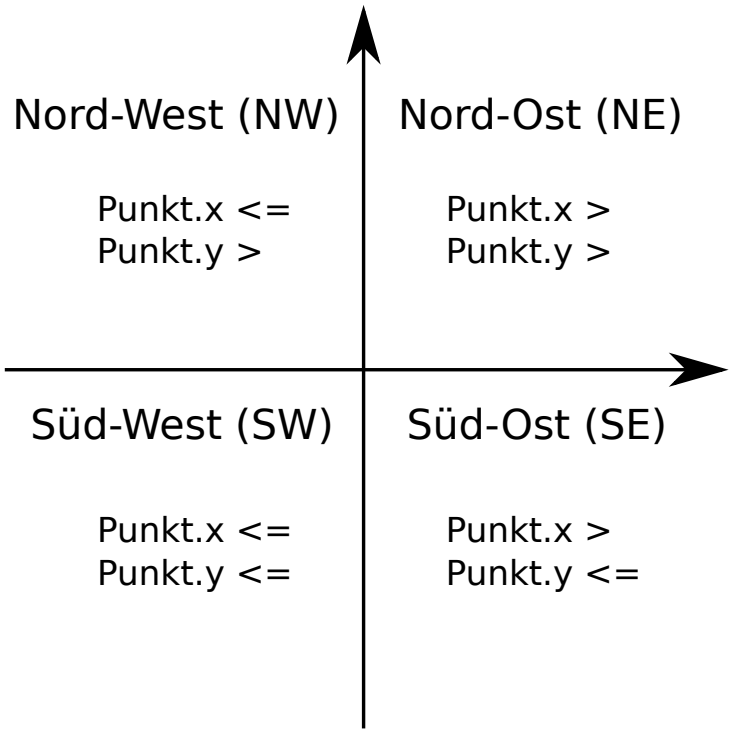

Abbildung 4.1: Die Quadranten eines quadtrees (zwei Dimensionen)

Natürlich kann ein Quadtree auch mehr als nur zwei Dimensionen unterstützen. Zu bedenken ist jedoch, dass sich die Anzahl der Quadranten mit jeder weiteren Dimension verdoppelt. Bei dreidimensionalen Geometrien wären es somit schon acht Quadranten, bei vier Dimensionen entsprechend sechzehn. Hierdurch entstehen gerade bei vielen Dimensionen sehr große Knoten mit entsprechend vielen nicht gesetzten Verweisen. Anzumerken ist noch, dass der Name quadtree bei mehr als zwei Dimensionen an sich nicht mehr korrekt ist, trotzdem jedoch regelmäßig anzutreffen ist. Korrekterweise handelt es sich bei

[1] Im Rahmen des kd-trees wurde kleiner und größer/gleich verwendet. Für den Quadtree hat sich aber eingebürgert genau die andere Variante zu verwenden.

drei Dimensionen um einen octree. Für mehr Dimensionen gibt es keine gebräuchlichen Namen.

Wichtig ist noch anzumerken, dass ein quadtree ähnlich dem kd-tree von sich aus nur Punktgeometrien unterstützt. Ähnlich wie beim kd-tree gibt es aber auch hier entsprechende Varianten, von denen eine im Abschnitt 4.7 vorgestellt werden soll.

Von einem gewissen Interesse sollte zudem sein, dass die Quadranten (im zweidimensionalen Raum) in der Regel als Rechtecke betrachtet werden. Zwar ergibt sich dies nicht aus der Definition selbst, da lediglich die Relation zu einem Schlüssen betrachtet wird. Somit wären zumindest alle außenliegenden Quadranten eines Baums in mindestens eine Richtung unbegrenzt. Es erleichtert aber die Betrachtung und die Implementierung, wenn aus dem vorliegenden Datensatz oder den momentan eingefügten Daten eine Bounding Box bestimmt und auf den Baum angewendet wird. Gleichwohl ist es nicht notwendig, dass zu einem Quadranten korrespondierende Rechteck im jeweiligen Knoten zu speichern, da sich dieses jeweils aus der Position im Baum ableitet (ausgehend von der gedachten Bounding Box).

4.2 Einfügen

Ähnlich dem kd-tree ist auch im quadtree das Einfügen eines Features eine relativ einfache Operation[2]. Ist der Baum noch leer, so genügt es einen Wurzelknoten zu erzeugen und als Schlüssel das Feature bzw. dessen Geometrie einzutragen. Existiert hingegen schon eine Wurzel, so muss eine passende Einfügeposition ermittelt werden. Dazu wird der Baum durchlaufen und bei jedem Knoten der Teilbaum (Quadrant) gewählt, in dem die Geometrie des Features relativ zum Schlüssel liegt. Wenn dem Quadranten noch kein Teilbaum zugeordnet ist, so ist die Einfügeposition gefunden. Nun wird ein neues Blatt erzeugt mit dem Feature (Geometrie) als Schlüssel und mit dem gefundenen Quadranten verknüpft.

In Abbildung 4.2 wird beispielhaft ein aus sechs Einfügeoperationen resultierender Baum dargestellt. Die Features wurden in alphabetischer Reihenfolge (A-F) nacheinander eingefügt. Im linken Bereich wird die Aufteilung der Fläche sowie die Position der Features gezeigt. Im rechten Teil ist der dazu passende Baum zu sehen. Deutlich zu erkennen ist die große Anzahl leerer Referenzen (nicht ausgefüllte Kreise).

Ebenfalls auffällig ist die Abhängigkeit der Baumstruktur von der Reihenfolge, in der die Features eingefügt wurden. Angenommen in dem Beispiel wäre B vor A eingefügt worden,

[2] *Samet*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling) (wie Anm. 3), S. 28.

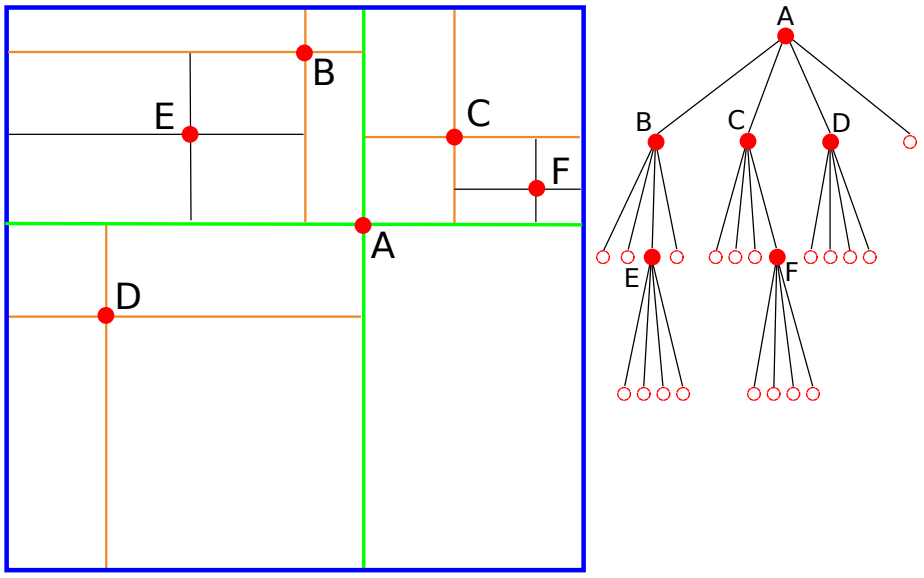

ABBILDUNG 4.2: Beispiel eines quadtrees

so würde hieraus ein völlig anderer Baum resultieren, wie in Abbildung 4.3 zu sehen ist. Nicht nur, dass die Wurzel eine andere wäre, auch die restlichen Knoten befinden sich z.T. an anderen Positionen im Baum. Zudem ist eine weitere Ebene hinzugekommen. Damit ähnelt der quadtree auch in dieser Hinsicht dem kd-tree.

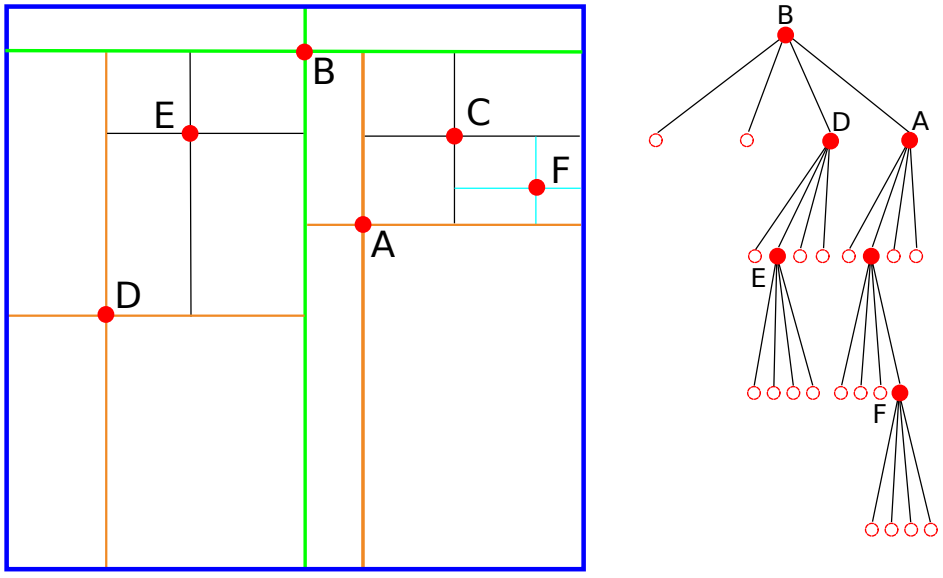

ABBILDUNG 4.3: Beispiel eines quadtrees mit veränderter Einfügereihenfolge

Neben dem sequenziellen Einfügen einzelner Features gibt es auch Vorschläge, komplette Datensätze auf einmal zu verarbeiten (statische Daten)[3]. Im Rahmen dieser Thesis wurde jedoch keiner der Ansätze weiter verfolgt.

4.3 Löschen

Prinzipiell gestaltet sich auch das Löschen von Features aus einem quadtree relativ einfach. Dazu muss die Position eines solchen Features im Baum ermittelt werden. Anschließend genügt es den Knoten zu löschen. Sollte dieser über Kindknoten verfügen, so sind alle dort referenzierten Features neu in den Baum einzufügen. Die Kindknoten sind anschließend ebenfalls zu entfernen.

Kritisch ist dabei jedoch anzumerken, dass die Effizienz des Verfahrens stark von der Position des Features innerhalb des Baums abhängt. Befindet es sich in einem Blatt, so ist der Löschvorgang sehr günstig. Steht es hingegen in der Wurzel, so muss der komplette restliche Baum neu eingefügt werden.

Der Vollständigkeit halber sei daher auf einen alternativen Algorithmus von Samet verwiesen, der jedoch im Kontext der Testumgebung nicht realisiert wurde. Dieser versucht, einen Ersatzknoten innerhalb des quadtrees zu finden, durch den die Baumstruktur möglichst wenig verändert werden muss[4]. Der Ansatz ist also ähnlich dem eines binären Suchbaums (bzw. kd-trees), jedoch ist es bei weitem schwieriger einen entsprechenden Kandidaten zu finden. Zudem gibt es Situationen, in denen eine größere Reorganisation nicht vermieden werden kann.

4.4 Suchen

Die Location- und Window-Query lassen sich mit geringem Aufwand in einem quadtree realisieren. Erstere wurde sogar bereits für das Einfügen und Löschen von Features benötigt. Letztere wiederum kann entsprechend aus dieser abgeleitet werden, wobei statt einer einzelnen Koordinate ein Bereich überprüft werden muss. Hierdurch kann es dazu kommen, dass mehrere Teilbäume durchlaufen werden müssen.

Auch die Nearest-Neighbor-Query kann entsprechend mit geringem Aufwand aus der Implementierung des kd-trees abgeleitet werden. Es genügt die Anpassung der Distanzfunktion, also der Bestimmung der Entfernung von der Ausgangsposition zu einem Knoten

[3] *Samet*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling) (wie Anm. 3), S. 30.
[4] Ebd., S. 31.

sowie die Berücksichtigung aller Quadranten statt nur des linken und rechten Teilbaums beim Ermitteln der Kinder.

4.5 Bewertung

Der quadtree ist ein Baum, der sehr viele parallelen zum kd-tree aufweist. So teilt er dessen Vorteil, die Einfachheit, aber ebenso auch dessen Nachteile. Er unterstützt nur Punktgeometrien, ist ebenso anfällig für das Degenerieren des Baums durch seine Abhängigkeit von der Einfügereihenfolge in Kombination mit der fehlenden impliziten Ausbalanciertheit. Gravierender ist noch das Problem der leeren Referenzen. Dafür ist er besser in der Lage den Raum zu unterteilen, da er alle Dimensionen gleichzeitig berücksichtigt und nicht an eine bestimmte Reihenfolge dieser gebunden ist. Dies kann aber auch als Nachteil gewertet werden, da auch immer alle Dimensionen betrachtet werden müssen.

Ein weiterer großer Nachteil ist wie bereits erwähnt das Löschen von Features. Insbesondere das Löschen von Knoten nahe der Wurzel hat bei größeren Bäumen entsprechend hohe Kosten. Da jedoch in der Regel die meisten Knoten bei einem Baum Blätter sind, ist davon auszugehen, dass der Worst Case entsprechend selten eintritt.

Hinsichtlich seiner Kosten ist bei Einfüge- und Suchoperationen (Window Query, Location Query) im Normalfall von $O(\log_4 N)$ bei N Knoten im Baum zu rechnen. Als zu erwartende Höhe wird im Durchschnitt $2c/d \ln N$ genannt, wobei empirisch $c = 4,31107$ ermittelt wurde[5].

4.6 bucket pr-quadtree

Nach Vorstellung des klassischen quadtrees soll nun auf eine Variante dessen eingegangen werden. Wie auch schon im Kontext der kd-trees ausgeführt, wäre eine solche wünschenswert, die mehrere Features in einem Knoten verwalten kann. Dies wiederum geht ein her mit der Forderung, den Raum und nicht die Objektmenge (Featuremenge) zu unterteilen. Verzichtet werden soll aber vorerst auf die Unterstützung von LineStrings und Polygons, auf die erst in Abbschnitt 4.7 eingegangen werden soll.

Bei der hier vorgestellten Variante handelt es sich um den pr-quadtree (point region quadtree)[6] in einer um Buckets erweiterten Form. Die Funktionsweise dieses Index kann

[5] *Samet*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling) (wie Anm. 3), S. 30.
[6] Ebd., S. 42.

mit wenig Aufwand aus dem klassischen quadtree abgeleitet werden. Da, wie bereits ausgeführt, der Raum selbst unterteilt werden soll, sind die Schlüsselwerte der Knoten entsprechend frei wählbar. Jedoch bietet es sich an, sie so zu wählen, dass alle Quadranten gleich groß sind. Wären dies nicht der Fall, so gäbe es das Risiko, dass die größeren Quadranten entsprechend größere Teilbäume bedingen würden. Dadurch würde der gesamte Baum entsprechend weniger ausbalanciert sein. Letztlich lohnt sich eine andere Wahl der Schlüsselelemente also nur, wenn entsprechendes Wissen über das Datenmaterial vorliegt. Im Falle von pr-quadtrees findet dies aber keine Berücksichtigung.

Da die Features (Geometrien) nun nicht mehr als Schlüsselelement benötigt werden, bietet es sich an, diese nur noch in den Blättern zu speichern. Hierdurch wird der Baum unabhängig von der Reihenfolge der Einfügeoperationen. Zudem ist so gewährleistet, dass ein Punkt nur in genau einem Knoten gespeichert sein kann und nicht mehr in allen Knoten auf dem Pfad zum Blatt. In den Blättern selbst können dann all jene Features referenziert werden, deren Geometrie genau innerhalb des dem Knoten zugeordneten Rechtecks liegen. Zu Berücksichtigen ist aber, dass ein solcher Knoten auch zu voll werden kann. Folglich wird eine entsprechende Split-Operation benötigt.

Durch die Unterteilung des Raums und der Forderung, dass die Quadranten eines Knotens gleich groß sind, stellt sich aber ein Problem, welches in dieser Form beim normalen quadtree nicht existiert hat. Es ist nun notwendig den Gesamtraum zu Beginn zu spezifizieren, also Festzulegen, welche maximale Ausdehnung dieser haben wird. Hieraus kann dann das dem Wurzelknoten zugeordnete Rechteck abgeleitet werden. Eine Alternative hierzu wäre, den Baum auch zur Wurzel hin wachsen zu lassen. Eine solche Arbeitsweise ist aber im pr-quadtree nicht vorgesehen.

An dieser Stelle ist noch einmal auf die Definition eines Quadranten im quadtree hinzuweisen (Abbildung 4.1). Diese ist selbstverständlich auch für den pr-quadtree gültig. Auf Rechtecke übertragen bedeutet das, dass die untere sowie linke Kante exklusive, die obere und die rechte Kante hingegen inklusive sind.

Abbildung 4.4 zeigt das bereits aus der Vorstellung des quadtrees bekannte Beispiel, angewendet auf einen pr-quadtree. Der Übersichtlichkeit halber wurde die maximale Anzahl an Features pro Blattknoten auf Eins gesetzt. Abweichend von der bisherigen Darstellung werden in der Abbildung die Blattknoten wegen ihrer besonderen Rolle als Rechtecke dargestellt.

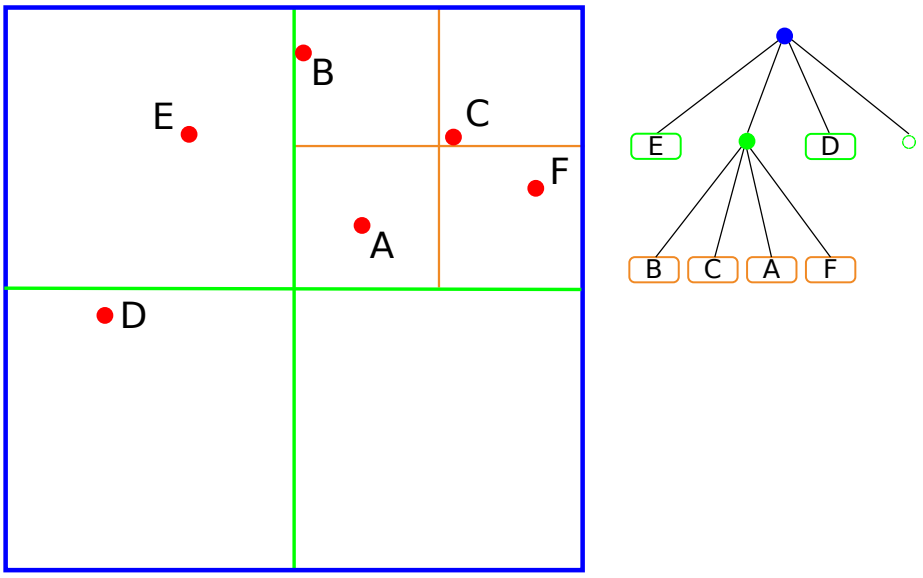

ABBILDUNG 4.4: Beispiel eines pr-quadtrees mit einem Feature pro Blattknoten

4.6.1 Einfügen

Ausgehend von der allgemeinen Beschreibung sollen nun, wie auch bei den anderen Indices, die einzelnen Operationen näher beleuchtet werden. Den Anfang bildet dabei die Einfügeoperation. Das grundlegende Verfahren ist, wie zu erwarten war, ähnlich dem des quadtrees. Falls keine Wurzel existiert, so wird ein neuer Blattknoten erzeugt und das einzufügende Feature dort eingesetzt. Falls hingegen bereits ein Wurzelknoten existiert, so muss eine passende Einfügeposition gesucht werden. Hierzu wird der Baum durchlaufen und jeweils der Quadrant gewählt, in dem der Punkt liegt. Die Suche terminiert, wenn einem solchen Quadranten noch kein Knoten zugeordnet wurde (leere Referenz) oder wenn ein Blattknoten erreicht wird.

Im Falle von ersterem ist nun ein neuer Blattknoten zu erzeugen. Anschließend wird das Feature in beiden Fällen dort einzutragen. Wenn nun hierdurch die Kapazität des Blattknotens überschritten wurde, so ist ein Split durchzuführen. Dazu wird der Blattknoten in einen regulären Knoten umgewandelt und anschließend alle enthaltenen Features über den beschriebenen Algorithmus in diesen eingefügt (welcher aus Sicht des Algorithmus in diesem Moment quasi als Wurzel fungiert). Während des Prozesses werden dann entsprechende neue Blattknoten angelegt. Ebenso kann es aber dabei auch zu neuen Splitoperationen kommen. Dies tritt immer dann ein, wenn alle Features relativ zum Rechteck des

Knotens sehr nahe zusammen liegen und eine Unterteilung von diesem nicht genügt um diese Menge auf mehrere Quadranten aufzusplitten[7].

4.6.2 Löschen

Die Löschoperation im pr-quadtree kann als direkte Umkehrung der Einfügeoperation implementiert werden. Auch hier ist zuerst die Position des zu löschenden Features im Baum zu suchen (identisch zur Einfügeposition). Anschließend wird dieses aus dem dabei gefundenen Blattknoten entfernt. Ist der Knoten nun leer so ist er zu entfernen. Dabei sollte jedoch der Elternknoten berücksichtigt werden, da dieser in der Folge ggf. nur noch einen Kindknoten besitzt. Ist dieser nämlich selbst ein Blattknoten so kann er ebenfalls entfernt werden. Der Elternknoten wird dann in ein Blatt umgewandelt und nimmt die übrig gebliebenen Features auf. Der Prozess wird als *collapsing* bezeichnet und stellt das Gegenstück zum *Split* beim Einfügen dar. Nicht durchgeführt werden kann er, wenn das einzige andere Kind des Elternknotens kein Blatt ist. In einem solchen Fall kann nicht sichergestellt werden, dass alle sich darunter befindlichen Features in einen Knoten hineinpassen. Dann wiederum wäre nämlich möglicherweise eine umfangreiche Reorganisation des Teilbaums notwendig, die entsprechend hohe Kosten verursachen würde.

4.6.3 Suchen

Die Suchoperationen Location Query, Window Query und Nearest Neighbor Query lassen sich mit überschaubarem Aufwand an den bucket pr-quadtree anpassen. Erstere wurde bereits in Grundzügen im Rahmen des Einfügens vorgestellt. Die Window Query kann wie üblich aus dieser abgeleitet werden. Die Nearest Neighbor Query hingegen benötigt einige kleinere Modifikationen und ähnelt dann stark der Implementierung im kdb-tree, jedoch ohne Berücksichtigung von Duplikaten, da diese im PR-Quadtree nicht vorkommen. Konkret ist dabei die Unterscheidung von Blattknoten (Features) und inneren Knoten zu realisieren sowie die Unterstützung von mehreren Features pro Blattknoten.

4.6.4 Bewertung

Zweifelsohne ist der bucket pr-quadtree bei größeren Datenmengen günstiger als ein klassischer quadtree. Durch die Unterstützung von Buckets ist die zu erwartende Höhe des

[7] Sollten mehrere Features mit gleicher Position eingefügt werden, so kann es zu einer Endlosschleife beim Split kommen. In diesem Fall muss der Blattknoten geeignet vergrößert werden um alle Features aufnehmen zu können. Im Rahmen der Testumgebung wurden wie auch beim kdb-tree Swap-Pages verwendet, wobei abweichend die letzte (Feature-)Referenz in einem solchen Fall auf die jeweilige Swap-Page zeigt.

Baums entsprechend geringer, was sich wiederum direkt positiv auf die Kosten aller Operationen auswirkt. Durch die Unabhängigkeit von der Reihenfolge der Einfügeoperationen ist die Gefahr eines Worst Case bei dynamischen Daten entsprechend geringer. Vielmehr ist das Szenario abhängig von der Verteilung der Daten selbst. Sind diese gleichmäßig verteilt, so tendiert der bucket pr-quadtree zur Ausbalanciertheit hinsichtlich der Höhe. Konzentrieren sie sich hingegen auf wenige kleinräumige Cluster ist der bucket pr-quadtree entsprechend ungünstig. Dabei sind vor allem zwei Aspekte zu nennen. So steigt erstens die Gesamthöhe des Baums entsprechend stark an. Dies wiederum erhöht die Kosten für alle Operationen und erhöht auch die Anzahl der leeren Referenzen signifikant. Gleichzeitig vergrößert sich zweitens auch die Differenz der Höhen der verschiedenen Teilbäume. Außerhalb der Cluster ist diese eher niedrig. In den Clustern, wie schon ausgeführt, hingegen sehr hoch. Dadurch ist es schwer vorherzusagen, welche Kosten eine Abfrage konkret besitzt. Folglich ist es abhängig von dem Datenmaterial, ob ein bucket pr-quadtree sinnvoll eingesetzt werden kann oder nicht.

4.7 mx-cif Quadtree

Die bis hierhin vorgestellten quadtrees unterstützen allein Punktgeometrien. Um nun aber für Geodaten universell einsetzbar zu sein, müssen auch LineStrings und Polygone indiziert werden können. Hierzu soll der mx-cif quadtree vorgestellt werden[8]. Der interessierte Leser sei darauf aufmerksam gemacht, dass es in der Literatur neben der hier vorgestellten noch eine zweite Variante des mx-cif quadtrees gibt, die unabhängig von dieser entwickelt wurde.

Der mx-cif quadtree unterscheidet sich im Ansatz von den beiden im Rahmen des kd-trees vorgestellten Ansätzen zur Indizierung von Nicht-Punktgeometrien. Gleichwohl setzt er genauso auf MBRs auf, indiziert also nicht die Geometrie direkt. Doch warum ist ein anderer Ansatz notwendig? Angenommen, man würde die Methode der repräsentativen Punkte übernehmen, so wären hierfür vier Dimensionen notwendig. Folglich hätte ein entsprechender quadtree sechzehn Quadranten pro Knoten. Allein die hierdurch bedingten leeren Referenzen sprechen gegen einen solchen Baum. Zudem ist es aber auch nicht ideal die Breiten- und Höheninformation eines MBRs in separaten Dimensionen zu speichern, wenn prinzipiell bereits die beiden regulären dazu in der Lage wären. Zumal die dadurch bedingte „Zersplitterung“ der Informationen entsprechende Suchoperationen (insbesondere die Nearest Neighbor-Operation) verkompliziert.

[8] *Gershon Kedem*: The quad-CIF tree: A data structure for hierarchical on-line algorithms, in: Proceedings of the 19th Design Automation Conference (DAC ’82), Piscataway, NJ, USA 1982, S. 352–357.

Die andere im Rahmen der kd-trees vorgestellte Variante, nämlich die Duplizierung der Features, wäre durchaus denkbar. Die dafür notwendigen Anpassungen beispielsweise im bucket pr-quadtree können leicht identifiziert werden. Allerdings verkomplizieren sich auch hier die meisten Operationen durch die Duplizierung, da eben immer damit gerechnet werden muss, dass ein Feature (deutlich) mehr als einmal im Baum vorkommt.

Der mx-cif quadtree hingegen verzichtet sowohl auf die Abbildung auf repräsentative Punkte als auch auf die Duplizierung von Features. Um dies zu erreichen muss man sich als Ausgangspunkt einen bucket pr-quadtree vorstellen. Soll darin ein durch einen MBR repräsentiertes Feature eingefügt werden, so würde man intuitiv ähnlich dem Einfügen von Punktgeometrien vorgehen. Beginnend mit der Suche nach einer passenden Einfügeposition stellt sich jedoch die Frage, nach welchen Kriterien eine einzige Position auszuwählen ist. Nimmt man die Blätter, so kann es vorkommen, dass diese bereits so feingranular den Raum unterteilen, dass der MBR nicht mehr vollständig hineinpasst. Wandert man nun aber den Baum hinauf, so findet man sehr wohl einen Knoten, der das Feature vollständig aufnehmen könnte[9]. Dieser Knoten kann gleichwohl auch erst die Wurzel sein. Formaler ausgedrückt wird ein Feature immer in dem Knoten referenziert, in dessen Rechteck seine Geometrie vollständig enthalten ist. Da dies allein aber auf mehrere Knoten zutreffen kann (für die Wurzel immer), muss zudem noch gefordert werden, dass seine Geometrie in keinem der Quadranten des Knotens vollständig enthalten ist. Abbildung 4.5 zeigt zur Verdeutlichung einen mx-cif quadtree mit fünf Geometrien. Auf die Darstellung der MBRs der Geometrien wurde der Übersichtlichkeit halber verzichtet, die Polygone wurden aber so entworfen, dass diese gedanklich leicht ergänzt werden können.

Aus dieser Konstruktionsregel lassen sich relativ schnell zwei Probleme ableiten, die entsprechend zu lösen sind. Erstens kann es passieren, dass in einem Knoten mehr Features referenziert sind, als dieser von sich aus aufnehmen kann. Da es aber keine Möglichkeit gibt, den Knoten zu teilen, ist es folglich notwendig diesen entweder zu vergrößern oder entsprechende „Auslagerungsknoten“ (Swap-Pages) zu schaffen, also Knoten, die ausschließlich dafür existieren, die überzähligen Features aufzunehmen. Durch diese entsteht eine Art von verketteter Liste. In der hier realisierten Implementierung viel die Wahl auf letzteres, da durch die Pagestruktur der Testumgebung eine beliebige Vergrößerung der Knoten nicht möglich ist (vgl. kdb-tree und bucket pr-quadtree).

Eine Folge hiervon ist, dass bei gleichmäßig verteilten Features auch sehr viele kleine in Knoten nahe der Wurzel referenziert werden. Konkret handelt es sich dabei um jene, die die Grenze zweier Quadranten schneiden. Da der mx-cif quadtree ursprünglich für den

[9] Es wird weiterhin vorausgesetzt, dass das Feature vollständig in dem beim Anlegen der Wurzel definierten Rechteck (Raum) liegt.

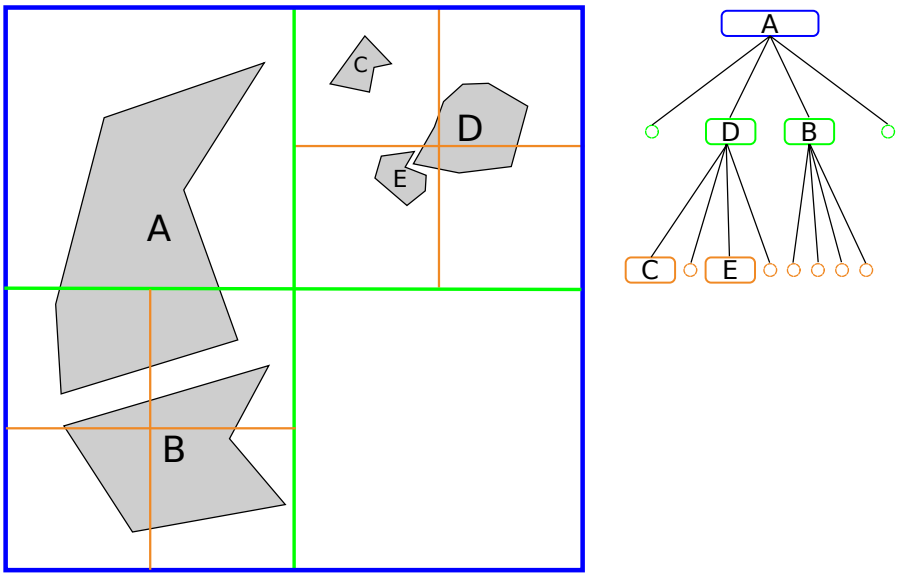

Abbildung 4.5: Beispiel eines mx-cif quadtrees

VLSI-Entwurf konzipiert wurde, bei dem gerade diese Konstellation der Regelfall der Fall ist, wird vorgeschlagen, die Features eines Knotens nicht einfach nur in einer Liste abzulegen. Stattdessen werden zwei binäre Suchbäume, einen für die X-Achse und einen für die Y-Achse konstruiert, in denen jeweils die entsprechenden Dimensionen der MBRs zusätzlich indiziert werden. Abbildung 4.6[10] zeigt beispielhaft die Indizierung von zwei MBRs entlang einer Achse. Da bei Geodaten jedoch nicht mit einer gleichmäßigen Verteilung zu rechnen ist, wurde in der hier realisierten Implementierung darauf verzichtet.

Das zweite Problem, das aus dem Konstruktionsregeln abgeleitet werden kann, ist das Einfügen sehr kleiner Geometrien. Als Extremfall sind hier insbesondere Punktgeometrien zu nennen. Wenn für eine solche Geometrie ein passender Knoten gesucht werden soll, so ist dieser folglich sehr Tief im Baum angeordnet. Im Falle von Punktgeometrien wäre es sogar gar nicht möglich einen passenden Knoten zu finden. Folglich muss eine maximale Tiefe des Baums vorab festgelegt werden. Die Suche nach einer passenden Einfügeposition terminiert dann spätestens wenn genau diese Tiefe erreicht wurde.

4.7.1 Einfügen

Der Algorithmus zum Einfügen von Features wurde bereits grob geschildert. Der erste Schritt ist wie auch bei den anderen quadtrees die Suche nach der Einfügeposition. Dabei wird, wie bereits dargelegt, der Knoten gesucht, in dessen Begrenzung die Geometrie

[10] *Kedem*: The quad-CIF tree: A data structure for hierarchical on-line algorithms (wie Anm. 8), S. 5.

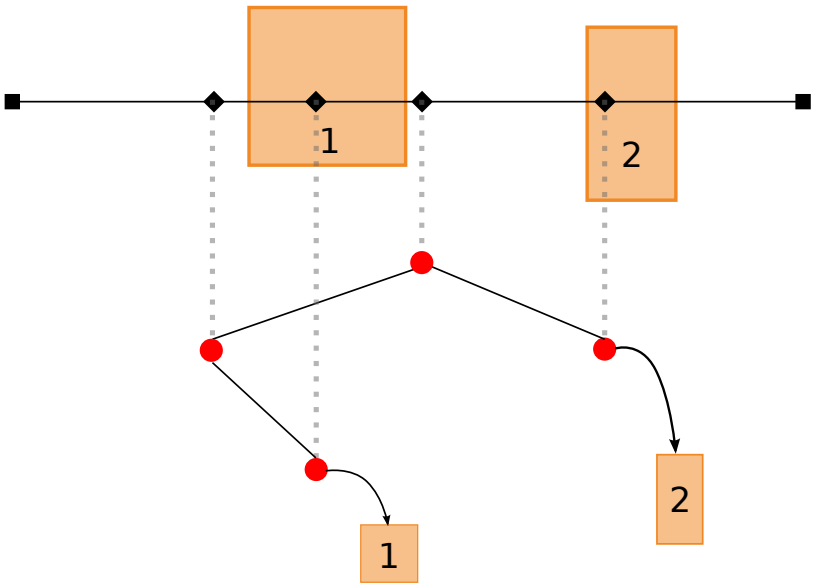

ABBILDUNG 4.6: Organisation der in einem Knoten abgelegten Features im MX-CIF

vollständig enthalten ist, der aber kein potentielles Kind besitzt, für das das gleiche gilt. Gleichzeitig ist noch die maximale Höhe des Baums zu berücksichtigen.

Folglich muss die Suche ausgehend von der Wurzel jeweils die einzelnen Quadranten hinsichtlich ihrer Geometrie in Relation zum Feature überprüfen. Überschneiden sich beide in Teilen so ist der momentane Knoten die Einfügeposition. Andernfalls wird der Teilbaum weiterverfolgt, in dessen Begrenzung die Geometrie vollständig enthalten ist. Ist der Teilbaum noch nicht erzeugt (leere Referenz), so ist ein neuer Blattknoten anzulegen. Anschließend ist der Vorgang entsprechend oft zu wiederholen, bis entweder ein passender Knoten gefunden oder die maximale Höhe erreicht wurde.

Abschließend muss das Feature nur noch an der entsprechenden Position referenziert werden, wobei darauf zu achten ist, dass die Kapazität des Knotens ausreicht. Falls notwendig ist der Knoten entsprechend zusätzliche Kapazität zu schaffen.

4.7.2 Löschen

Um ein Feature aus dem mx-cif quadtree zu löschen ist dieses wie üblich zuerst zu suchen. Ist der Knoten gefunden, in dem es enthalten ist, so wird es dort entfernt. Sollten binäre Suchbäume zur Organisation der Features innerhalb eines Knotens eingesetzt werden, so muss ferner auf deren Konsistenz nach dem Löschvorgang geachtet werden.

Falls nach dem Entfernen des Features der Knoten nun leer ist, d.h. weder Kindknoten noch Features enthalten sind, so ist dieser zu entfernen. Der Schritt wird anschließend rekursiv auf den jeweiligen Elternknoten angewandt, bis ein Knoten erreicht wurde, der nicht als leer klassifiziert werden kann (*collapsing*).

4.7.3 Suchen

Wie bereits beim bucket pr-quadtree lassen sich auch hier die Suchalgorithmen mit entsprechend geringem Aufwand anpassen. Abweichend ist im mx-cif quadtree zusätzlich jeweils noch der MBR des referenzierten Features gegen die Suchgeometrie zu prüfen[11], statt wie bisher direkt die Punktgeometrie.

Analog ist auch die Nearest Neighbor Query anzupassen, wobei es sich hier empfiehlt die Distanz direkt gegen die echte Geometrie der Features zu bestimmen, da es bei dieser Abfrage prinzipiell keine Kandidaten gibt. Alternativ wäre ein zweistufiger Prozess denkbar, bei dem zuerst die Distanz auf Basis des MBRs bestimmt wird und anschließend bei einem erneuten „Treffer“ erst die eigentliche Distanz ermittelt wird.

4.7.4 Bewertung

Der mx-cif quadtree ist sicherlich geeignet, Nicht-Punktgeometrien zu indizieren. Voraussetzung ist aber, dass die maximale Höhe sinnvoll gewählt wird. Ist diese zu hoch, so besteht die Gefahr, dass die meisten Features weit von der Wurzel entfernt in den Blättern referenziert werden (entsprechend kleine Geometrien angenommen). Damit erhöht sich gleichzeitig die Laufzeit aller Operationen beträchtlich. Ist die Höhe hingegen zu gering, so unterteilt der Baum den Raum nicht genug. In diesem Fall kann die Verwendung von binären Suchbäumen eine gewisse Linderung bedeuten, letztlich bleibt es aber eine eher ungünstige Situation.

Wenig geeignet erscheint der Index hingegen für Punktgeometrien. Da diese immer in den Blättern gespeichert werden, sind die Kosten in einem solchen Fall direkt abhängig von der gewählten Maximalhöhe. Diese wird allerdings global für den gesamten Baum festgelegt.Folglich passt sich dieser nicht an die Verteilung der Geometrien innerhalb der Datenmenge an. Hier ist daher davon auszugehen, dass der bucket pr-quadtree entsprechend bessere Eigenschaften zeigt.

[11] Es sei daran erinnert, dass der Index lediglich Kandidaten zurückliefert. Die abschließende (rechenintensive) Prüfung gegen die eigentliche Geometrie geschieht an anderer Stelle.

Kapitel 5

r-trees

5.1 Einführung

Bereits im Kontext der kdb-trees wurden kurz die r-trees erwähnt. Vielleicht sind diese dem geneigten Leser auch schon an anderer Stelle begegnet, denn es handelt sich um die bekannteste Gruppe räumlicher Indices. In vielen Datenbanksystemen übernehmen sie die Indizierung von Geodaten.

Doch was sind r-trees bzw. auf deutsch R-Bäume? Die bisher vorgestellten Indices haben, unabhängig davon, ob sie die Objektmenge oder den Raum unterteilt haben, einige Gemeinsamkeiten. So gab es gewissermaßen keinen nicht indizierten Bereich. Entweder weil die Unterteilung in bestimmte Richtungen keine Begrenzung besessen hat oder weil bereits beim Anlegen der Wurzel der maximale Bereich (quasi die Welt aus Sicht des Index) definiert wurde. Dieser Bereich wurde dann nur noch in den Teilbäumen jeweils weiter unterteilt.

Ebenfalls waren sowohl der kd-tree als auch der quadtree ursprünglich nur für Punktgeometrien konzipiert. Andere Geometrientypen wurden erst nachträglich in entsprechende Varianten unterstützt, was jedoch eigene Probleme und Unzulänglichkeiten bedeutet hat.

Der r-tree, genauer der guttman r-tree[1] wie die ursprüngliche Variante zwecks Unterscheidbarkeit genannt wird, wurde hingegen von Anfang an für die Indizierung von MBRs, also Rechtecken konzipiert. Daher rührt auch der Name, rectangle-tree. Ausgangspunkt ist dieses mal nicht der binäre Suchbaum sondern der B-Baum, wie er aus Datenbanksystemen bekannt ist und wie er bereits im kdb-tree verwendet wurde.

[1] *Antonin Guttman*: R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD international conference on Management of data (SIGMOD '84), Boston, Massachusetts 1984, S. 47–57.

Um nun den Grundgedanken des r-trees zu verstehen nehmen wir an, es liegt eine Menge von Geometrien in Form von MBRs vor. Diese Geometrien sind müssen nicht gleichmäßig verteilt sein und können sich überlappen, womit das gleiche auch für die MBRs gilt. Abbildung 5.1, Schritt 1 zeigt beispielhaft einen solche Geometriemenge. Die Idee des R-Baums ist nun diese MBRs wieder zu geeigneten Gruppen zusammenzufassen. Diesen kann dann wieder ein MBR zugeordnet werden. Die Basis bilden dann aber nicht die Geometrien selbst sondern deren MBRs[2] (Abbildung 5.1, Schritt 2). Nun gibt es aber noch ggf. mehrere solchen Gruppen-MBRs. Um einen Baum zu erhalten müssen diese wiederum mit dem selben Prozess zusammengefasst werden. Dieser wird letztlich so lange angewendet, bis nur noch ein MBR übrig bleibt, womit die Wurzel gefunden ist. Für die Geometriemenge aus Abbildung 5.1 ist dies in Schritt 3 zu sehen.

Gut zu erkennen ist bei diesem Vorgehen die Verwandtschaft zum B-Baum. Damit einher geht insbesondere die Ausbalanciertheit hinsichtlich der Höhe, bedingt durch das wachsen des Index zur Wurzel hin. Ebenfalls deutlich wird, dass sich alle Features (Geometrien) auf der untersten Ebene in den Blättern befinden. Alle inneren Knoten gruppieren lediglich noch die Featuremenge.

Zusammenfassend ergeben sich die folgenden Eigenschaften bzw. Bedingungen für einen r-tree[3]:

1. Jeder Blattknoten referenziert zwischen m und M Features.
2. Zu jedem im Blattknoten referenzierten Feature muss auch der zugehörige MBR gespeichert werden.
3. Jeder Knoten, der kein Blatt ist, besitzt zwischen m und M Kinder. Ausgenommen hiervon ist der Wurzelknoten.
4. Zu jeder Referenz auf einen Kindknoten ist dessen MBR zu speichern.
5. Die Wurzel hat mindestens zwei Kinder oder ist ein Blattknoten.
6. Alle Blätter sind auf der selben Ebene im Baum.

Gleichwohl gilt es bei der Konstruktion eines r-trees darüber hinaus noch eine Reihe von Dingen zu beachten. Es genügt nicht für einen effizienten Index beliebig die MBRs einer Ebene zu gruppieren. Zur Verdeutlichung dieses Sachverhalts sei noch einmal auf Abbildung 5.1 verwiesen. Konkret stellt sich hier die Frage, wieso A und B sowie C, D und E jeweils gruppiert wurden. Zwar ist einem Menschen diese Wahl beim Blick auf die Grafik

[2] Prinzipiell ist es egal, ob die Geometrien oder ihre MBRs zur Bildung eines Gesamt-MBRs herangezogen werden. Für den konzeptionellen Ansatz ist aber der Weg über letzteres von großer Bedeutung.
[3] *Guttman*: R-trees: a dynamic index structure for spatial searching (wie Anm. 1), S. 48.

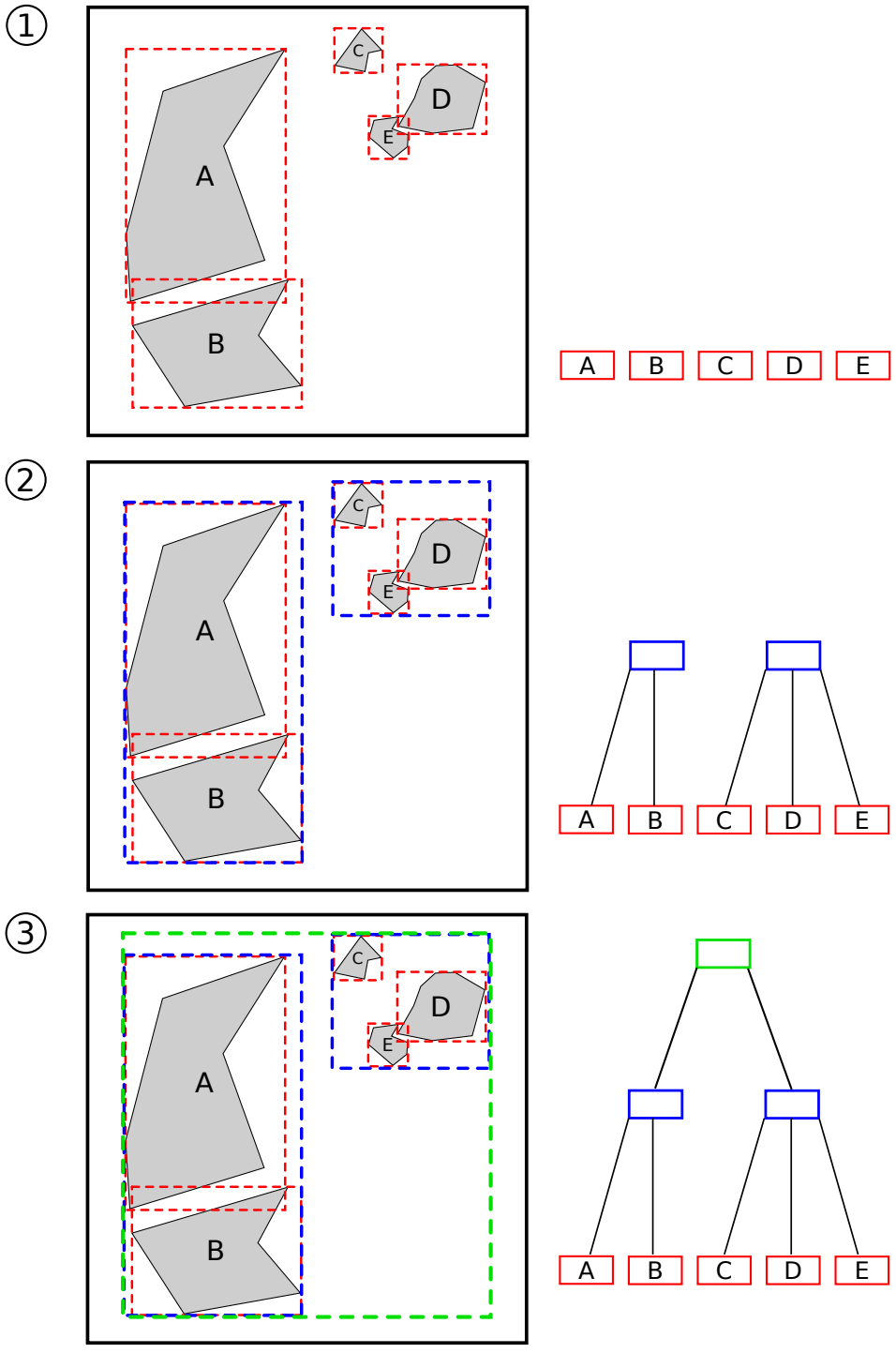

ABBILDUNG 5.1: Konstruktion eines r-trees aus einer Menge von Geometrien

relativ einleuchtend, doch was würde einen Computer daran hindern E der ersten statt der zweiten Gruppe zuzuordnen? Und welche Strategie sollte angewendet werden, wenn nun eine hypothetische Geometrie F an der unteren rechten Ecke hinzugefügt werden würde? Was geschieht generell, wenn eine Gruppe (Knoten) durch Einfügen zu viele Elemente ($> M$) enthält, als dass man sie noch verwalten könnte? Ebenso ist die Gegenrichtung relevant, nämlich die Reaktion auf einen durch Löschen zu leeren Knoten ($< m$).

Die Beantwortung all dieser Fragen ist keineswegs trivial und hat zu einer Vielzahl von Varianten des r-trees geführt. Selbst im ursprünglichen Entwurf von Guttman wurden gleich drei verschiedene Ansätze für die Splitoperation vorgestellt (siehe Abschnitt 5.2). Eine umfangreichere Analyse der verschiedenen Möglichkeiten sowie der relevanten Kriterien wird in Abschnitt 5.7 im Rahmen der Diskussion des r*-trees präsentiert. Abschnitt 5.6 wiederum zeigt einen Lösungsansatz für rein statische Daten, den str-tree. Zu Anfang soll aber nun erst einmal auf die konkreten Operationen im guttman r-tree eingegangen werden.

5.2 Einfügen

Ausgehend von der Einführung ist bereits deutlich geworden, dass der Prozess des Einfügens im r-tree deutlich komplexer ist als bei den bisher vorgestellten Indices. Daher empfiehlt es sich zu Anfang die Operation gedanklich in zwei Teile zu zerlegen, die bereits aus den vorherigen Kapiteln deutlich geworden sind: der Ermittlung der Einfügeposition und dem Einfügen selbst.

Beginnend mit ersterem stellt sich dieses mal die Frage, wie überhaupt eine solche Position ermittelt werden kann. Bei allen bisherigen Bäumen war immer (in gewisser Weise) der gesamte Raum abgedeckt. Es war also vor dem Beginn des Suchvorgangs klar, dass es einen Knoten im Baum geben wird, der die Position oder den Bereich auch abdeckt. Im r-tree hingegen kann es sein, dass noch kein solcher Knoten existiert. Als Beispiel sei angenommen, dass Straßen in NRW indiziert werden sollen. Wenn der Einfügeprozess nun Stadt für Stadt vorgeht, in der Reihenfolge Dortmund, Gummersbach, Köln usw und nun die erste Straße von Gummersbach eingefügt wird, so ist davon auszugehen, dass auch die Wurzel den betreffenden Bereich noch nicht abdeckt. Das gleiche gilt selbstverständlich erst recht für die einzelnen Teilbäume.

Da es wenig ökonomisch ist, nun jeweils eine neue Wurzel inklusive Teilbaum für das gerade einzufügende Feature zu erstellen - von der Problematik abgesehen, dass alle Blätter auf der selben Ebene liegen sollen - muss folglich eine andere Lösung existieren. Umgesetzt im r-tree ist, dass statt eines neuen Knotens immer der nächst beste im jeweiligen

Teilbaum herangezogen wird. Im Idealfall handelt es sich um einen Knoten, der bereits die betreffende Stelle abdeckt. Hier ist aber zu beachten, dass es mehrere solche Knoten geben kann, da sich die MBRs überlappen können! Guttman schlägt daher zwei Kriterien vor, nach denen ein Knoten (Teilbaum) für ein neu einzufügendes Feature F ausgewählt werden soll:

1. Wähle den Knoten, dessen MBR am wenigsten vergrößert werden muss um F aufzunehmen.
2. Falls mehrere Knoten ermittelt wurden: Wähle den mit der geringsten Fläche (MBR).

Der Suchvorgang beendet, wenn ein Blattknoten erreicht wurde. In diesen wird nun in einem zweiten Schritt das Feature in Form der Referenz sowie des MBRs eingefügt. Anschließend gilt es zu prüfen, ob der Knoten nach wie vor maximal M Elemente enthält. Ist die Bedingung erfüllt, so sind nun die MBRs aller Knoten vom Blatt bis zur Wurzel hin zu aktualisieren, da sich ihre Beschaffenheit durch den Einfügevorgang verändert haben kann. Dazu wird der Baum von Blatt (Einfügeposition) aus zur Wurzel hin durchlaufen und jeweils der neu berechnete MBR in den Elternknoten geschrieben.

Problematischer hingegen ist es, wenn die Bedingung nicht mehr eingehalten wird, also der Blattknoten zu viele Elemente beinhaltet. In einem solchen Fall muss ein Split durchgeführt werden. Das Ziel ist dabei wie auch bisher den Knoten in zwei neue zu zerlegen, doch gibt es beim r-tree noch einige weitere Dinge zu berücksichtigen. Hierzu genügt es sich die Frage zu stellen, wie denn die Elemente genau auf die beiden neuen Knoten aufzuteilen sind. Dies bedingt dann wiederum ihre MBRs. Als Kriterium für einen guten Split schlägt Guttman vor, dass die Gesamtfläche beider neuer Knoten minimiert werden sollte[4]. Argumentativ rührt dies daher, dass beim Suchen im Baum mittels Bounding Box (Window Query) die Wahrscheinlichkeit, dass ein Knoten berücksichtigt werden muss, sinkt, wenn dessen Fläche klein ist.

Doch ist es keineswegs Trivial die Aufteilung zu finden, bei der die Gesamtfläche minimal ist. Um dies zu berechnen müssten letztlich alle Kombinationen geprüft werden, was bei $M+1$ Features ungefähr 2^{M-1} Kombinationen bedeutet. Die Laufzeit ist folglich als exponentiell einzuordnen. Es werden daher neben dem als *Exhaustive Algorithm* bezeichneten Variante noch zwei weitere von Guttman präsentiert, eine mit quadratischer und eine mit linearer Laufzeit.

Ersterer, der *Quadratic-Cost Algorithm*, versucht eine hinsichtlich der Gesamtfläche möglichst gute Kombination zu ermittelt. Es ist jedoch nicht garantiert, dass diese auch

[4] *Guttman*: R-trees: a dynamic index structure for spatial searching (wie Anm. 1), S. 51.

die beste ist. Ausgangspunkt sind dabei die beiden Elemente, die zusammen in einem MBR die meiste Fläche verschwenden würden ($Gesamtfläche - Feature1.Fläche - Feature2.Fläche$ ist maximal). Diese bilden jeweils den Ausgangspunkt für die beiden neuen Knoten $K1$ und $K2$.

Nun müssen die verbleibenden Elemente verteilt werden. Dazu wird iterativ jeweils ein geeignetes Element aus der verbliebenen Menge ausgewählt. Kriterium ist, dass die Differenz des Flächenzuwachses von $K1$ und $K2$ bei Hinzunahme dieses Elements maximal ist. Logischerweise wird das ausgewählte Element zu dem Knoten hinzugefügt, für den es den geringeren Flächenzuwachs bedeutet. Der Split ist dann beendet, wenn alle Elemente verteilt wurden oder wenn alle verbleibenden einem Knoten zugeordnet werden müssen, damit dieser die Mindestanzahl von m erreicht.

Das letzte Verfahren, der *Linear-Cost Algorithm*, ist ähnlich der eben vorgestellten. Er unterscheidet sich allerdings in der Wahl der beiden Startknoten sowie in dem Kriterium zur Auswahl des jeweils nächsten Elements. Als Ausgangspunkt werden die beiden Knoten gewählt, die am weitesten von einander entfernt sind[5]. Anschließend werden die verbliebenen Elemente in beliebiger Reihenfolge ausgewählt und jeweils zu dem Knoten hinzugefügt, für den dies den geringeren Flächenzuwachs bedeutet[6].

Laut Guttman sind der *Quadratic-Cost Algorithm* und der *Linear-Cost Algorithm* als gleichwertig zu betrachten[7]. Es wird daher keine konkrete Empfehlung hinsichtlich der Wahl gegeben. Lediglich vom *Exhaustive Algorithm* wird abgeraten. Im Rahmen der Testumgebung wurde der *Linear-Cost Algorithm* gewählt.

Nach Durchführung eines Splits kann es natürlich wiederum auch im Elternknoten zu einem Split kommen. In diesem Fall ist dort entsprechend analog zu verfahren. Letztlich muss der Baum bis zur Wurzel durchlaufen werden. Auf jeder Ebene ist dann zumindest der MBR analog dem normalen Einfügen anzupassen und bei bedarf ein Split durchzuführen.

[5] Zur Bestimmung des Paares werden die Dimensionen jeweils einzeln betrachtet und für jede Dimension das am weitesten entfernte Pärchen gewählt. Anschließend wird die Distanz durch eine Division durch die Gesamtbreite der Dimension normalisiert und das Paar gewählt, dessen normalisierte Distanz am größten ist.

[6] Für den Fall, dass der Flächenzuwachs gleich ist, wurde in der Testumgebung eine Erweiterung realisiert: Zuerst wird der Knoten mit der geringeren Fläche gewählt, falls dies nicht genügt der Knoten mit weniger Kindern.

[7] *Guttman*: R-trees: a dynamic index structure for spatial searching (wie Anm. 1), S. 56.

5.3 Löschen

Das Löschen eines Features aus dem Index erfolgt in drei Schritten: dem Suchen des Features im Baum, dem Löschen sowie zum Schluss dem Bereinigen. Der Suchvorgang gestaltet sich so, dass ausgehend von der Wurzel jeweils der MBR des Features F_{mbr} gegen den MBR jedes Kindes F_{mbr} geprüft wird. Ist F_{mbr} vollständig in K_{mbr} enthalten, so ist der jeweilige Teilbaum rekursiv weiter zu durchsuchen. Zu beachten ist, dass es nicht genügt, nur den ersten „getroffenen“ Teilbaum weiter zu durchlaufen, da, wie bereits geschildert, sich die MBRs überlappen können.

Wurde beim Suchvorgang ein Blatt erreicht, so ist dort zu prüfen, ob das Feature enthalten ist. Falls ja, so ist die Suche beendet und das Feature wird aus dem Knoten entfernt. Im Anschluss ist der Baum entsprechend zu bereinigen, da sich die MBRs geändert haben können. Zudem kann es passieren, dass der Blattknoten nun weniger als m Einträge besitzt. Dazu wird der Index vom Blattknoten aus zur Wurzel hin durchlaufen und auf jeder Ebene der entsprechende Knoten im Pfad geprüft. Hat ein solcher weniger als m Einträge, so werden alle noch verbliebenen zur Menge Q hinzugefügt und der Knoten entfernt. Bleibt der Knoten hingegen bestehen, so ist dessen MBR zu berechnen in im zugehörigen Elternknoten zu aktualisieren.

Nachdem die Wurzel erreicht wurde (die Eliminierung sowie Aktualisierung des MBRs finden hier logischerweise keine Anwendung mehr) werden alle Elemente, die während des Durchlaufs in Q abgelegt wurden erneut in den Baum eingefügt. Dabei ist aber zu berücksichtigen, auf welcher Ebene sie entfernt wurden, da es sich bei dem Elementen sowohl um Features als auch um Teilbäume handeln kann. Als letzter Schritt ist abschließend noch zu prüfen, ob die Wurzel nun nur noch exakt einen Kindknoten besitzt. Ist dies der Fall, so wird dieser zur neuen Wurzel. Die alte Wurzel ist zu entfernen.

5.4 Suchen

Die Suche hinsichtlich der Window und Location Query wurde bereits in Grundzügen im Rahmen des Einfügens und Löschens beschrieben, wenn auch jeweils angepasst auf die konkrete Operation. Für die normale Suche sieht der Ablauf derart aus, dass ausgehend von der Wurzel jeweils die MBRs aller Kinder gegen die Suchgeometrie (Bounding Box, Position) zu prüfen sind. Alle Teilbäume, mit deren MBR es eine Schnittmenge gibt, sind rekursiv weiter zu analysieren. Bei jedem Blattknoten ist entsprechend analog die Schnittmengen zu den MBRs aller referenzierten Features zu festzustellen. Besteht eine

Schnittmenge, so ist das Feature in die Menge der Kandidaten K aufzunehmen. Nach Durchlauf aller relevanten Teilbäume ist K als Ergebnis der Suche zurückzugeben.

Einfacher hingegen ist die Anpassung des Nearest Neighbor-Algorithmus. Die Implementierung erfolgt dabei ähnlich der des bucket pr-quadtrees und des mx-cif quadtrees. Der Abstand wird dabei für Knoten basierend auf ihrem MBR berechnet. Für Features ist keine Änderung notwendig, die Berechnung kann weiterhin gegen die Geometrie erfolgen. Zu berücksichtigen ist, dass Features nur im Blattknoten vorkommen.

5.5 Bewertung

Der Vorteil des r-trees ist, dass er im Gegensatz zu den bisher vorgestellten Indices nicht nur für Punktgeometrien konzipiert wurde. Dadurch kann insbesondere im Vergleich zum four-dimensional kd-tree (Abschnitt 3.6) und kdb-tree (Abschnitt 3.7) von einem besseren Laufzeitverhalten ausgegangen werden. Hinzu kommt, dass der Baum durch seine Verwandtschaft mit dem B-Tree von sich aus hinsichtlich seiner Höhe ausbalanciert ist. Damit stellt sich das Problem der Worst-Case Szenarien (zu linearen Listen degenerierte Bäume) der bisher vorgestellten Bäume nicht. Gleichwohl besitzt der r-tree andere Schwachpunkte (Worst-Case Fälle).

Ein weiterer Vorteil ist, dass anders als bisher, der freie Raum nicht indiziert wird. Im günstigen Fall kann somit bereits im Wurzelknoten eine Abfrage entsprechend beantwortet werden. Gleichzeitig stellt aber gerade dieser Punkt auch einen ersten Nachteil dar, da Einfügeoperationen hierdurch entsprechend komplexer sind. Es ist jedoch davon auszugehen, dass in der Praxis hier der Vorteil überwiegt, da von mehr Such- als Einfügeoperationen unterstellt werden können.

Ebenfalls problematisch ist, dass der klassische guttman r-tree nicht für statische Daten konzipiert wurde. Zwar lassen sich solche Daten auch über entsprechend viele einzelne Einfügeoperationen verarbeiten, dabei sind aber zwei Dinge zu bedenken. Erstens ist die Struktur des Baums von der Einfügereihenfolge abhängig. Es gibt folglich optimale und weniger optimale Reihenfolgen. Es ist aber nicht im Vorfeld abzuschätzen, welche zu wählen ist. Im ungünstigen Fall erhält man dann einen Index, bei dem es viele Knoten mit einem hohen Anteil an freier Fläche gibt oder bei dem sich viele Knoten auf einer Ebene überlappen. Beides erhöht dann im Betrieb unnötig die Laufzeit von Suchoperationen. Das zweite zu bedenkende Problem ist, dass nicht davon ausgegangen werden kann, dass die Mehrzahl der Knoten M Einträge besitzen wird. Möglich ist vielmehr auch der Worst-Case, dass lediglich m Einträge pro Knoten vorhanden sind. Für statische Daten wäre aber vielmehr ein Wert von M, also 100% Füllgrad, günstig, da hiermit die Anzahl

der vorhandenen Knoten auf das notwendige Maß reduziert werden würde. Hierdurch reduziert sich gleichzeitig die Anzahl der zu durchlaufenden Pfade wie auch die Höhe des Baums

Kritisch zu prüfen ist, ob Punktgeometrien sinnvoll in einem r-tree indiziert werden können. Sicherlich kann die zu erwartende Laufzeit nicht als unangemessen betrachtet werden, doch ist hier primär der Vergleich zu den bisher vorgestellten Indizes von Relevanz, da diese ursächlich für Punktgeometrien konzipiert wurden. Eine entsprechende Erwartungshaltung gegenüber diesen ist daher sicherlich nicht vollkommen abwegig.

5.6 str-tree

Als populäre Lösung für das Problem der statischen Daten im Kontext des r-trees hat sich der str-tree etabliert[8]. Der str-tree, die Kurzform für Sort Tile Recursive-Tree, begegnet dabei den bereits geschilderten Schwierigkeiten mit einem relativ einfachen Verfahren. Allerdings ist bei diesem darauf hinzuweisen, dass auch nur statische Daten berücksichtigt werden können, da alle Features in einem einzigen Schritt eingefügt werden. Spätere separate Einfügeoperationen sind zwar theoretisch denkbar, jedoch praktisch mit sehr vielen Splits verbunden, da lediglich der bereits bekannte Algorithmus aus dem klassischen r-tree zur Verfügung steht.

Die Grundidee des Verfahrens ist, die MBRs aller Geometrien in möglichst gleichmäßigen Kacheln zu gruppieren. Die Kacheln haben dann idealerweise exakt M Einträge, was jedoch nur möglich ist, wenn die Anzahl der Features ein Vielfaches von M beträgt. Der Prozess wird entsprechend auf den Kacheln wiederholt, bis nur noch eine übrig bleibt, die dann als Wurzel fungiert.

Es stellt sich allerdings die Frage, nach welchen Kriterien eine solche Gruppierung durchgeführt werden kann. Sicherlich bestünde die Möglichkeit, alle Kombinationen durchzuprobieren, um dann die Beste weiterzuverwenden. Doch ist dieser Ansatz für größere Datenbestände sicherlich nicht praktikabel. Im str-tree erfolgt dies vielmehr in zwei einzelnen Schritten. Ausgangspunkt ist, wie schon dargelegt, eine Menge von MBRs. Abbildung 5.2, Schritt 1 zeigt eine solche Menge mit insgesamt $N = 27$ MBRs. Ziel ist es nun, diese Menge in Knoten zu je M Elementen zu gruppieren, wobei im Beispiel $M = 3$ ist.

Zunächst werden die Schwerpunkte der MBRs bestimmt und anschließend alle entsprechend ihres x-Werts sortiert (Schritt 2). Anschließend werden diese im nächsten Schritt

[8] *Scott T. Leutenegger/Jeffrey M. Edgington/Mario A. Lopez*: STR: A simple and efficient algorithm for R-tree packing, Techn. Ber., 1997.

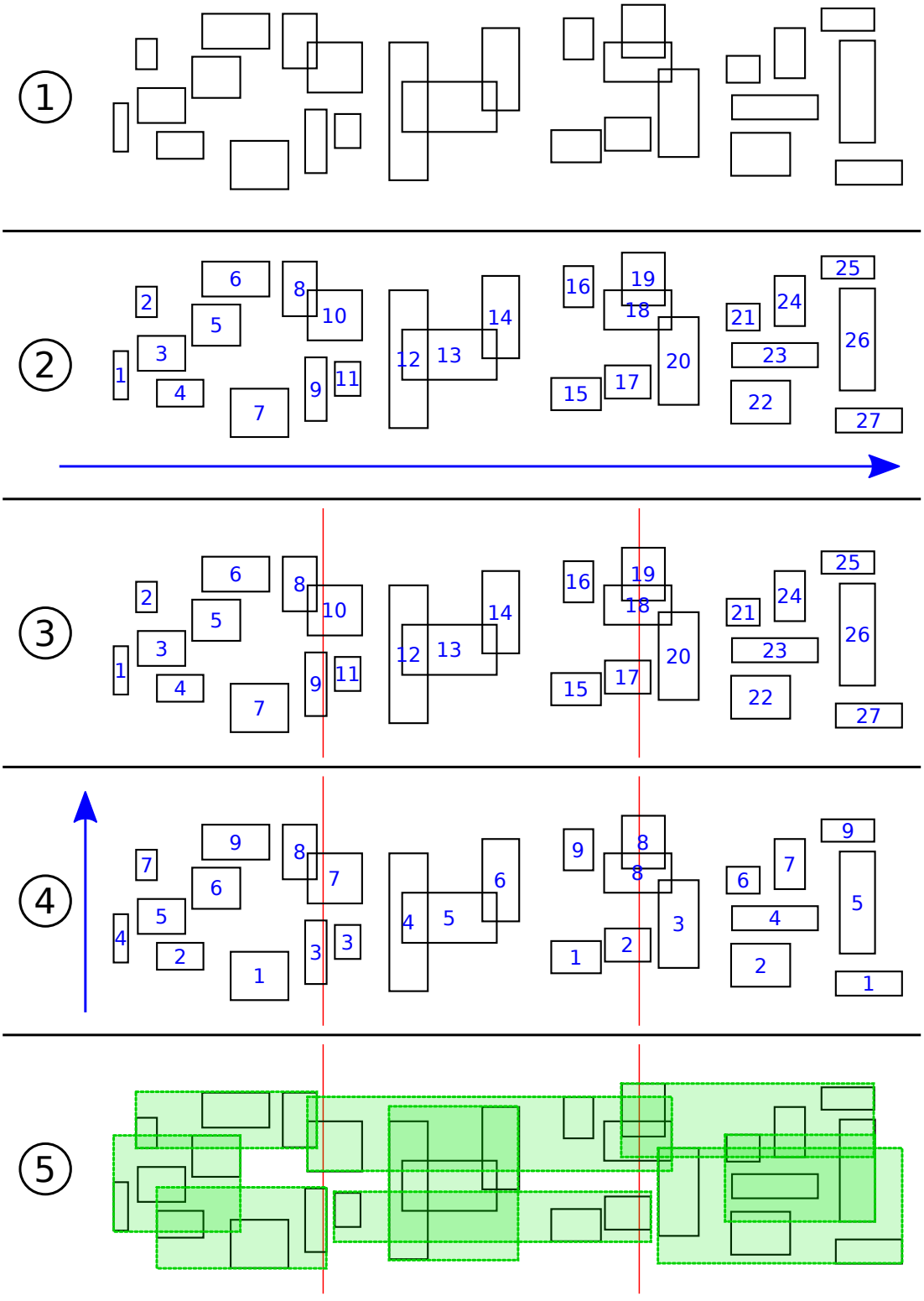

Abbildung 5.2: Konstruktion eines STR-Trees aus einer Menge von Geometrien

(Schritt 3) in $S = \left\lceil \sqrt{\lceil \frac{N}{M} \rceil} \right\rceil$ Slices zu je $S \cdot M$ aufeinander folgenden MBRs eingeteilt (entsprechend der Sortierung). Der letzte Slice kann unter Umständen weniger Elemente beinhalten. Für das Beispiel ergibt sich $S = 3$, folglich enthält jeder Slice in etwa $S \cdot M = 9$ Elemente. Anschließend werden nun alle MBRs innerhalb der Slices erneut sortiert, dieses Mal jedoch entsprechend des y-Werts ihres Schwerpunkts. Das Ergebnis ist dabei in Schritt 4 zu sehen. Als letztes folgt Schritt 5, in dem innerhalb eines Slices nun jeweils M aufeinander folgende Elemente zu einem Knoten gruppiert werden, woraus dann S Knoten pro Slice resultieren, insgesamt also $S \cdot S$ Knoten. Hierdurch ergibt sich auch die bereits erwähnte gleichmäßige Verteilung, da entlang beider Achsen jeweils gleich viele Knoten vorhanden sind.

Wie bereits dargelegt, werden diese Schritte nun so oft wiederholt (Rekursion), bis M oder weniger Elemente übrig bleiben, welche dann in einen einzigen Knoten gruppiert werden können. Dieser Knoten ist dann die Wurzel des Baums. Gut erkennbar ist nun auch, warum der Baum den Namen Sort Tile Recursive-Tree trägt. Hier ist letztlich die Abfolge der einzelnen Arbeitsschritte beschrieben, also das Sortieren der MBRs, das Kacheln dieser sowie die rekursive Anwendung der beiden Schritte bis man die Wurzel erhalten hat.

5.6.1 Bewertung

Der str-tree ist für statische Daten ein sehr interessanter Index, der mit relativ geringem Aufwand implementiert werden kann. Es ist von einer besseren Abfragegeschwindigkeit auszugehen als beim r-tree. Von Bedeutung ist auch, dass der Baum mit relativ geringen Kosten erzeugt werden kann, wobei hier primär die Wahl des Sortieralgorithmus von Bedeutung ist. Damit ist er auch für statische Daten mit geringer Lebensdauer von Bedeutung.

Gleichwohl birgt das Vorgehen auch gewisse Schwächen. Durch die Wahl des Schwerpunkts kann es bei heterogenen Datensätzen zu größeren Überlappungen kommen. Dies kann Ansatzweise bereits in der Abbildung 5.2, Schritt 5 in den Slices zwei und drei gesehen werden. Dies wiederum hat dann Auswirkungen auf die Laufzeit entsprechender Abfrageoperationen[9]. Interessanterweise ist das Problem bei Punktgeometrien gegenstandslos, da hier der Schwerpunkt des MBRs identisch zur Geometrie selbst ist. Es kann folglich nicht der Fall auftreten, dass es durch die Ausmaße der Geometrie zu einer Überlappung mit anderen Knoten kommt. Lediglich das berühren zweier Knoten ist hier denkbar, wenn zwei Punkte den selben y-Wert besitzen und beide MBRs entsprechend

[9] *Leutenegger/Edgington/Lopez*: STR: A simple and efficient algorithm for R-tree packing (wie Anm. 8), S. 8.

geformt sind[10]. Es kann also von einer deutlich besseren Unterstützung von Punktgeometrien ausgegangen werden.

5.7 R*-Tree

Eine Weiterentwicklung des r-trees für dynamische Daten stellt der r*-tree dar[11]. Kern dieses Index ist ein mittels experimenteller Messungen verbessertes Einfüge- bzw. Splitverfahren. Ausgangspunkt der Verbesserungen war die Definition einige Kennzahlen. Von diesen wurde angenommen, dass sie signifikante Auswirkungen auf die Kosten insbesondere von Suchoperationen haben[12]:

O1 *Die von einem MBR eines Knotens abgedeckte Fläche sollte minimal sein.* Gemeint ist hier die Fläche, die direkt vom MBR abgedeckt wird, die also nicht zugleich Teil eines Kindknotens oder eines Features ist (toter Raum). Es wird davon ausgegangen, dass eine Minimierung dazu führt, dass die Auswahl der relevanten Teilbäume höher im Baum durchgeführt werden kann. Dies wiederum wirkt sich positiv auf die Abfragegeschwindigkeit aus. Das Kriterium entspricht dem bereits in r-tree verwendeten Optimierungsziel.

O2 *Die Überlappungen zwischen den MBRs von Knoten der gleichen Ebene sollte minimal sein.* Ebenso wie **O1** wird hierdurch die Auswahl der relevanten Teilbäume verbessert, wodurch weniger Pfade durchlaufen werden müssen.

O3 *Der Umfang der MBRs von Knoten sollte minimal sein.* Der Umfang ist hier als die Summe aller Seitenlängen eines MBRs zu verstehen. Dies wiederum führt dazu, dass all jene MBRs bevorzugt werden, die annähernd quadratisch sind. Hintergrund für diese Forderung sind zwei Annahmen. Ersten wird davon ausgegangen, dass Window Queries häufig mit nahezu quadratischen Bounding Boxen durchgeführt werden. Sind nun die MBRs ähnlich geformt, so werden diese Suchabfragen besser unterstützt, da weniger Pfade durchlaufen werden müssen. Zweitens führen nahezu quadratische MBRs auch zu einer besseren Baumstruktur, da sich diese einfacher gruppieren lassen als ungünstig (länglich) geformte Rechtecke.

[10] Falls deutlich mehr als M Geometrien den selben y-Wert haben, kann es vorkommen, dass ein Knoten keine Ausdehnung hinsichtlich der y-Achse besitzt und dann vollständig auf der Hülle eines anderen Knotens liegt. Somit ist er korrekterweise sogar vollständig in diesem enthalten.

[11] *Norbert Beckmann* u. a.: The R*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD international conference on Management of data (SIGMOD '90), Atlantic City, New Jersey, United States 1990, S. 322–331.

[12] Ebd., S. 323.

O4 *Knoten sollten möglichst voll sein.* Hier gilt das bereits im Rahmen des str-trees genannte, nämlich dass insbesondere hierdurch die Höhe des Baums reduziert werden kann.

Gleichwohl lassen sich nicht alle Kennzahlen gleichermaßen berücksichtigen, denn wie leicht zu erkennen ist beeinflussen sich diese gegenseitig. So ist es zum Erreichen von **O1** oder **O2** notwendig, die Knoten flexibler zu gestalten, also weniger Elemente hinzuzufügen (**O4**) oder den MBR ungleichmäßiger zu formen (**O3**). Es besteht folglich ein Zielkonflikt zwischen den verschiedenen Kennzahlen. Im Rahmen der Entwicklung des r*-trees wurde daher an verschiedenen Stellen jeweils überprüft, welche Kennzahl die besten Ergebnisse, also die geringsten Kosten, bewirkt und der Algorithmus entsprechend modifiziert.

Die eigentlichen Modifikationen am Einfügealgorithmus lassen sich in drei Teilbereiche zerlegen. Erstens die Auswahl der Einfügeposition, zweitens die Splitoperation und drittens dem Zusammenspiel beider. Letzteres umfasst auch neu hinzugekommene Ausführungspfade, die so nicht in der Vorlage vorhanden sind. Doch soll zunächst wie Wahl der Einfügeposition behandelt werden. Hier wird nun unterschieden zwischen der Auswahl eines inneren Knotens, welche unverändert gemäß **O1** durchgeführt wird, sowie der Auswahl eines Blattknotens, bei der nun entsprechend der experimentellen Ergebnisse **O2** das Auswahlkriterium darstellt[13]. Dies bedeutet, dass der Blattknoten als Einfügeposition ausgewählt wird, der durch Hinzunahme des neuen Features F am wenigsten Überlappungsfläche hinzugewinnt. Es wird aber darauf hingewiesen, dass dies nur mit quadratischer Laufzeit berechnet werden kann, da jeder Knoten mit jedem anderen innerhalb des selben Elternknotens verglichen werden muss. Die Autoren schlagen daher noch ein zweites Verfahren für große Werte für M vor, dass näherungsweise den besten Kandidaten ermittelt. Dazu werden zuerst alle Knoten aufsteigend ihrer Flächenzunahme bei Integration von F sortiert. Anschließend werden nur noch die ersten A Knoten gegen die Gesamtmenge geprüft. A ist geeignet zu wählen[14].

Der zweite Teilbereich, die Split-Operation, unterscheidet sich signifikant von den drei Varianten des klassischen r-trees. Ausgangspunkt für alle Schritte des Splits ist die Bildung von Verteilungen zu je zwei Gruppen. Dazu werden für jede Achse die MBRs zuerst entsprechend ihres unteren, dann ihres oberen Wertes sortiert. Anschließend werden für die hieraus resultierenden sortierten Mengen je $M - 2m + 2$ Verteilungen der insgesamt $M + 1$ Elemente erzeugt. Die k-te Verteilung bestimmt sich dadurch, dass die erste Gruppe die ersten $(m - 1) + k$ Einträge enthält. Die zweite Gruppe umfasst entsprechend den Rest. Abbildung 5.3 visualisiert dies.

[13] *Beckmann* u. a.: The R*-tree: an efficient and robust access method for points and rectangles (wie Anm. 11), S. 325.

[14] Da die Knotengröße in der Testumgebung einen niedrigen Wert aufweist wurde auf die Implementierung dieser Variante verzichtet.

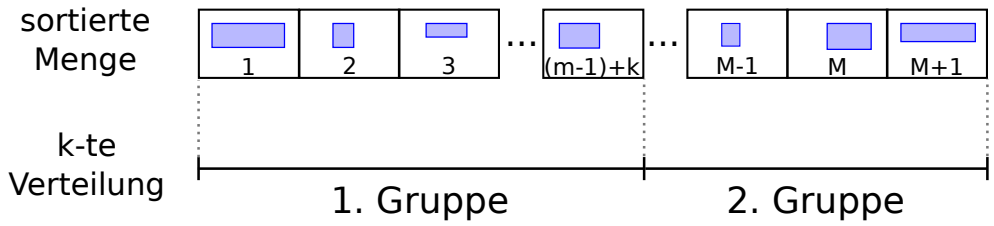

ABBILDUNG 5.3: r*-tree: Bildung der k-ten Verteilung aus einer sortierten Menge

Für jede Verteilung werden nun je drei Kennzahlen ermittelt, die Aussagen dazu erlauben, in wie weit sie in den verschiedenen Schritten des Algorithmus genutzt werden können. Diese Kennzahlen lassen sich aus den bereits geschilderten allgemeinen (**O1**-**O4**) ableiten, wobei **O4** im Kontext des Splits, wie leicht zu erkennen, keine Rolle spielt.

Fläche $area(mbr(erste\ Gruppe))+area(mbr(zweite\ Gruppe))$, also die Summe der Flächen der durch beide Gruppen jeweils aufgespannten MBRs.

Umfang $umfang(mbr(erste\ Gruppe)) + umfang(mbr(zweite\ Gruppe))$, also die Summe des Umfangs der durch beide Gruppen jeweils aufgespannten MBRs.

Überlappung $area(mbr(erste\ Gruppe) \cap mbr(zweite\ Gruppe))$, also die Fläche der Schnittmenge der durch beide Gruppen jeweils aufgespannten MBRs.

Ausgehend hiervon wird nun der folgende Algorithmus zum Durchführen eines Splits vorgeschlagen. Im ersten Schritt ist die Achse zu bestimmen, zu der orthogonal die Teilung erfolgt. Anschließend wird für die Achse die geeignetste Verteilung gewählt. Die hierbei entstandenen beiden Gruppen bilden dann die aus dem Split resultierenden neuen Knoten.

Um die Achse auszuwählen werden zuerst für jede alle Verteilungen gebildet. Nun wird für pro Achse die Summe der Umfangswerte S berechnet. Anschließend die für das weitere vorgehen gewählt, deren S am kleinsten ist. Um nun auf der gewählten Achse die beste Verteilung zu ermitteln ist wiederum für jede Verteilung der Überlappungswert zu berechnen und jene zu wählen, deren Wert am niedrigsten ist. Falls es kein eindeutiges Ergebnis gibt, so ist für alle hieraus ermittelten Kandidaten noch der Flächenwert zu bestimmen und wiederum die Verteilung zu wählen, deren Wert am geringsten ist.

Neben dem Split verfügt er r*-tree aber noch über eine weitere Operation um zu volle Knoten zu behandeln: die *forced reinsertion*. Hier hinter verbirgt sich nichts weiter als

das das Entfernen von einem Teil des Knoteninhalts in Verbindung mit dem anschließenden erneuten Einfügen eben dieses in den Baum. Als Grund für das Vorgehen sei daran erinnert, dass der r-tree abhängig von der Einfügereihenfolge ist, also der Baum entsprechend von der Abfolge der einzelnen Operationen bzw. der dabei betroffenen Datensätze jeweils eine andere Struktur annimmt. Dadurch kann es zu Degenerierungserscheinungen kommen. Insbesondere durch wiederholte Einfüge- und Löschoperationen in relativ begrenzten Regionen kann es passieren, dass dort die Knoten ungünstige Form annehmen (vgl. Kennzahlen **O1**-**O3**). In solchen Fällen können auch die jeweils ergriffenen Maßnahmen bei Einfüge- und Splitoperationen nur noch wenig Abhilfe schaffen. Daher wird für den r*-tree vorgeschlagen, alternativ komplette Teile neu einzufügen, in der Hoffnung, dass die Knoten anschließend eine bessere Geometrie aufweisen. Entsprechende experimentelle Daten scheinen dies zu untermauern[15].

Um nun geeignete Kandidaten für eine reinsertion zu ermitteln wird für jedes Element des Knotens die Distanz d vom Mittelpunkt des MBRs zum Mittelpunkt des Knotens bestimmt. Anschließend werden alle Elemente entsprechend d in absteigender Reihenfolge sortiert. Geeignete Kandidaten sind nun die ersten p Elemente dieser Liste, wobei $p = 0,3 \cdot M$ vorgeschlagen wird[16]. Anschließend werden die Kandidaten entfernt und die MBRs der Knoten wie üblich aktualisiert. Beim Einfügen ist dann noch die Reihenfolge zu bedenken, in der die Elemente wieder in den Baum integriert werden. Es wird empfohlen dies entsprechend d, jedoch in aufsteigender Reihenfolge, durchzuführen (*close reinsertion*).

Es bleibt darauf hinzuweisen, dass durch die *forced reinsertion* natürlich wiederum zu volle Knoten entstehen können. Aus diesem Grund sollte das Verfahren pro Ebene nur einmal pro (globaler) Einfügeoperation durchgeführt werden. In allen weiteren Fällen ist dann der reguläre Split anzuwenden. Ebenso sei noch einmal deutlich gemacht, dass dieses Verfahren keineswegs nur auf Blattknoten angewendet wird. Beim erneuten Einfügen ist folglich zu berücksichtigen, auf welcher Ebene die Elemente entfernt wurden, da sie letztlich auch auf exakt dieser wieder eingefügt werden müssen.

Nun können alle bisher besprochenen Teile zu einer geeigneten Einfügeoperation verknüpft werden. Ausgangspunkt ist wie üblich die Suche nach einer geeigneten Einfügeposition. Dies erfolgt nach dem geschilderten Verfahren. Im nächsten Schritt wird dann das neue Feature im Zielknoten referenziert. Sollten danach mehr als M Elemente im Knoten vorhanden sein, so ist entweder eine *forced reinsertion* oder ein Split durchzuführen, abhängig davon, ob für die momentane Ebene bereits eine *forced reinsertion* durchgeführt wurde. Falls ein Split gewählt wurde so ist der Prozess ggf. rekursiv auf die Elternknoten

[15] *Beckmann* u. a.: The R*-tree: an efficient and robust access method for points and rectangles (wie Anm. 11), S. 326.
[16] Ebd., S. 327.

anzuwenden. Zum Abschluss sind dann noch die MBRs entsprechend zu aktualisieren, so dass diese mit dem Inhalt der Knoten korrespondieren.

5.7.1 Bewertung

Zweifelsohne sind die Algorithmen des r*-trees deutlich komplexer als die des klassischen r-trees. Insbesondere die Split-Operation fällt hier auf. Als Kosten sind hier zwei Sortierungen zu je $O(MlogM)$ zu nennen. Der restliche Vorgang beschränkt sich auf die Berechnung von $2(2(M-2m+2))$ Umfangswerten pro Achse und $2(M-2m+2)$ Überlappungswerten[17]. Es ist jedoch zu berücksichtigen, dass die Split-Operation seltener angewendet wird, da die *forced reinsertion* das bevorzugte Verfahren darstellt.

Augenscheinlich scheint die Wahl der Kennzahlen für die einzelnen Schritte sinnvoll zu sein. Es ist daher zu erwarten, dass der r*-tree allgemein für dynamische Daten bessere Eigenschaften aufweist als der r-tree. Gleichwohl ist die Auswahl für den durchschnittlichen Anwendungsfall getroffen worden. Es kann also sein, dass für bestimmte Szenarien eine andere Wahl besser gewesen wäre (Geodaten). Genauso wenig geeignet wie der r-tree ist er hingegen für statische Daten, da hier, wie bereits ausgeführt, andere Verfahren benötigt werden (siehe Abschnitt 5.6).

[17] *Beckmann* u. a.: The R*-tree: an efficient and robust access method for points and rectangles (wie Anm. 11), S. 326.

Kapitel 6

Vergleich

Im letzten größeren Abschnitt dieser Thesis soll nun ein Vergleich zwischen den bisher vorgestellten Indices gemacht werden. Von Bedeutung sind dabei sowohl die grundlegenden Eigenschaften der Bäume als auch die durch die Testumgebung ermittelten experimentellen Daten. Das Kapitel gliedert sich dabei in zwei Teile. Zu Anfang soll auf die theoretische Seite eingegangen und die bisher im Rahmen der einzelnen Indices genannten Eigenschaften gegenüber gestellt werden. Im zweiten Teil wird dann ein Vergleich auf Basis der ermittelten Messwerte angestellt.

6.1 Vergleich der Eigenschaften

Die im Rahmen der Thesis vorgestellten Indices unterscheiden sich in einer Vielzahl an Eigenschaften. Dies war, wie bereits in der Einleitung genannt, auch ein Ziel bei der Auswahl. Eine der Eigenschaften ist sogar Bestandteil des Titels dieser Arbeit: die Unterscheidung zwischen statischen und dynamischen Geodaten. Wie in den vorherigen Kapiteln geschildert gibt es Indices, die für statische Daten und solche die für dynamische Daten geeignet sind. Zudem gibt es noch einige, die zwar für dynamische Geodaten entwickelt wurden, aber durch ihre Eigenschaften auf eine gewisse Eignung für statische Geodaten aufweisen.

Neben diesem für die Thesis sehr grundlegenden Charakteristikum können noch die folgenden weiteren Eigenschaften als relevant betrachtet werden:

Unterstütze Geometrietypen Es kann zwischen solchen Bäumen unterschieden werden, die ausschließlich für Punktgeometrien entwickelt wurden, solchen die sowohl

für Punktgeometrien als auch für LineStrings und Polygone geeignet sind und solche die nur für letzteres sinnvoll verwendet werden können. Abhängig von den zu indizierenden Geodaten muss folglich aus einer anderen Menge von Indices gewählt werden.

Reihenfolgenabhängigkeit Die Struktur einiger vorgestellter Bäume ist abhängig von der Reihenfolge der Einfüge- und Löschoperationen ist. Andere wiederum weisen unabhängig von der Reihenfolge am Ende immer den gleichen Aufbau auf. Diese Eigenschaft ist vor allem dann von Bedeutung, wenn wenige Änderungen (Manipulationsoperationen) durchgeführt werden. Ist nämlich der Aufbau abhängig von der Reihenfolge, so kann von günstigen und weniger günstigen Reihenfolgen ausgegangen werden. Je statischer die Daten umso länger kann sich eine ungünstige Reihenfolge auf die Geschwindigkeit auswirken. Je dynamischer die Daten umso eher kann es von Vorteil sein, dass der Index „flexibel reagieren“ kann.

Unterteilungsart Die vorgestellten Bäume unterscheiden sich darin, ob sie die Objektmenge oder den zu Grunde liegenden Raum selbst unterteilen. Die Fragestellung ist dabei eng mit der **Reihenfolgenabhängigkeit** verknüpft, wenn auch nicht gleich zu setzen. Liegt eine Unterteilung des Raums vor, so ist dies ein Indiz dafür, dass die Struktur des Baums unabhängig von der Reihenfolge der Operationen ist.

Behandlung freier Bereiche Es existieren Indices, die freie Bereiche nicht genauso als Teil des Index betrachten wie belegte. Dem gegenüber stehen die, bei denen freie Bereiche erst in den Blattknoten erkennbar sind oder durch nicht „ausgebildete“ Teilbäume signalisieren (z.B. bucket pr-quadtree). Die Eigenschaft ist dahingehend von Bedeutung, dass bei ungleichmäßig verteilten Daten in Verbindung mit Location und Window Queries eine frühzeitige Identifizierung freier Flächen von Vorteil ist.

Ausbalanciertheit Es kann zwischen Bäumen unterschieden werden, die inhärent hinsichtlich der Höhe ausbalanciert sind und solchen, bei denen die Höhe der Teilbäume stark variieren kann bis hin zur Degenerierung zu einer linearen Liste. Die Eigenschaft beeinflusst also deutlich den Best- und Worstcase bei Suchoperationen.

Unterstützung von Buckets Einige der vorgestellten Bäume unterstützen Buckets, also die Speicherung einer wählbaren Anzahl an Elementen in einem Knoten. Andere hingegen bieten nur die Speicherung von einer genau festgelegten (niedrigen) Anzahl an Features und/oder Kindern an. Dabei ist ersteres von Vorteil, wenn größere Datenmengen indiziert werden sollen, da mehr Elemente pro Knoten auch eine geringere Höhe des Baums bedeutet.

Speicherort der Features Die vorgestellten Indices können unterteilt werden in solche, die Features lediglich in Blattknoten referenzieren und solchen, die dies auch in inneren Knoten erlauben. Bei einer großen Höhe des Baums kann es von Vorteil sein, wenn die Features nur in den Blättern referenziert werden, insbesondere wenn eine größere Anzahl an Features in einem Knoten referenziert werden kann. Hierzu ist zu bedenken, dass bei genügend Höhe eine entsprechend große Anzahl an Features geprüft werden muss, wenn diese auch in inneren Knoten abgelegt werden können. Viele dieser Features müssten aber gar nicht geprüft werden, wenn sie nur in Blattknoten vorkommen würden, da sie sehr weit vom Suchbereich entfernt liegen können.

Es existieren also eine Reihe von Möglichkeiten, die vorgestellten Indices an Hand ihrer grundlegenden Eigenschaften zu unterscheiden. Zur Verdeutlichung zeigt Tabelle 6.1 noch einmal eine Zusammenfassung. Betrachtet man nun allein die Tabelle, so könnte man vermuten, dass der kdb-tree sowie alle r-trees für statische und/oder dynamische Geodaten eine gute Eignung aufweisen. Mit Einschränkungen kann dies auch für den mx-cif quadtree und den bucket-pr quadtree vermutet werden. Allein für den kd-tree, den four-dimensional kd-tree und den quadtree ist nicht von einer Eignung jenseits kleinerer Datenmengen auszugehen.

Doch handelt es sich dabei lediglich um Vermutungen, die so in der Praxis zwar zutreffen können, jedoch nicht müssen. Daher ist es an dieser Stelle nun sinnvoll, die Ergebnisse der einzelnen Tests zu zu betrachten dann unter Berücksichtigung beider Seiten entsprechende Schlüsse zu ziehen.

6.2 Vergleich der Messergebnisse

Die Präsentation der Messergebnisse gliedert sich in drei Teile, jeweils entsprechend der Art der zu Grunde liegenden Daten. Zu Anfang soll auf die Ergebnisse bei Verwendung von Punktgeometrien eingegangen werden. Daran schließt sich die Behandlung von Nicht-Punktgeometrien an. Den letzten Abschnitt bildet die Vorstellung der Ergebnisse aus den Tests mit Bewegungsdaten. Diese Gliederung ergibt sich daraus, dass wie bereits Eingangs erwähnt, abhängig von der Art der Geometriedaten, bestimmte Indices nicht verwendet werden können.

Vor Präsentation der Messergebnisse ist es noch sinnvoll, zuerst auf die Bedingungen einzugehen, unter denen die Daten erhoben wurden. Sofern anwendbar, wurden für die Indices $M = 8$ Kinder festgelegt. Im praktischen Einsatz mag abhängig vom Index und

	kd-tree	**four-dimensional kd-tree**	**kdb-tree**
Statisch/Dynamisch	Beides	Beides	Dynamisch
Geometrietypen	Nur Punkte	Keine Punkte	Beides
Reihenfolgeabh.	Ja	Ja	Ja
Unterteilungsart	Objekte	Objekte	Raum
Freie Bereiche Teil des Baums	Ja	Ja	Ja
Ausbalanciertheit	Nein	Nein	Ja
Buckets	Nein	Nein	Ja
Speicherort	Überall	Überall	Nur Blätter

	quadtree	**bucket pr-quadtree**	**mx-cif quadtree**
Statisch/Dynamisch	Dynamisch	Beides	Beides
Geometrietypen	Nur Punkte	Nur Punkte	Beides
Reihenfolgeabh.	Ja	Nein	Nein
Unterteilungsart	Objekte	Raum	Raum
Freie Bereiche Teil des Baums	Ja	Ja	Ja
Ausbalanciertheit	Nein	Nein	Nein
Buckets	Nein	Ja	Ja
Speicherort	Überall	Nur Blätter	Überall

	rtree	**str-tree**	**r*-tree**
Statisch/Dynamisch	Dynamisch	Statisch	Dynamisch
Geometrietypen	Beides	Beides	Beides
Reihenfolgeabh.	Ja	n.a.	Ja
Unterteilungsart	Objekte	Objekte	Objekte
Freie Bereiche Teil des Baums	Nein	Nein	Nein
Ausbalanciertheit	Ja	Ja	Ja
Buckets	Ja	Ja	Ja
Speicherort	Nur Blätter	Nur Blätter	Nur Blätter

Tabelle 6.1: Grundlegende Eigenschaften/Charakteristika der vorgestellten Indices

Index	Größe (byte)	Zusammensetzung
kd-tree	29	$Splitachse + Ref_{Links} + Ref_{Rechts} + Punkt_2 + Ref_{Feature}$
four-dimensional kd-tree	45	$Splitachse + Ref_{Links} + Ref_{Rechts} + Punkt_4 + Ref_{Feature}$
kdb-tree	327	$Knotentyp + Splitachse + MBR + M + M \cdot (Ref_{Kind} + MBR) + Ref_{Swappage}$
quadtree	36	$Punkt_2 + Ref_{Feature} + Ref_{nw} + Ref_{ne} + Ref_{sw} + Ref_{se}$
bucket pr-quadtree	162	$Knotentyp + max(Ref_{nw} + Ref_{ne} + Ref_{sw} + Ref_{se},\ M + M \cdot (Ref_{Feature} + Punkt_2))$
mx-cif quadtree	309	$Knotentyp + Ref_{nw} + Ref_{ne} + Ref_{sw} + Ref_{se} + M + M \cdot (Ref_{Feature} + MBR) + Ref_{Swappage}$
r-tree (alle Varianten)	290	$Knotentyp + M + M \cdot (Ref_{Kind} + MBR)$

TABELLE 6.2: Knoten- bzw. Pagegrößen bei $M = 8$ Kindern.

den konkreten Gegebenheiten ein anderer Wert gewählt werden. Zwecks der Vergleichbarkeit viel die Entscheidung aber auf eine für alle Bäume einheitliche Größe. Zudem wurde der Wert niedrig gewählt, um einen entsprechend großen Baum auch bei einer Datenmenge durchschnittlichen Umfangs zu erzielen. Es ist folglich bei denjenigen Indices ohne Unterstützung von Buckets von einer gewissen Bevorteilung auszugehen und bei Betrachtung der Ergebnisse zu berücksichtigen.

Aus der Wahl für M kann nun die Größe der Knoten/Pages entsprechend ermittelt werden. Tabelle 6.2 bietet hierzu eine Aufstellung. In der Erläuterung der Zusammensetzung werden dabei die folgenden Symbole verwendet: Ref bezeichnet eine Referenz/Zeiger auf ein anderes Element/Feature und ist mit vier Byte veranschlagt. $Punkt_d$ bezeichnet einen Punkt der Dimension d und wird pro Dimension mit acht Byte kodiert. MBR wiederum kennzeichnet einen aus zwei Punkten zu je zwei Dimensionen zusammengesetzten MBR. Alle anderen Werte sind auf jeweils ein Byte beschränkt.

Alle Tests wurden auf einem Computer mit einem Intel Core i7 - 860 als CPU und einer Samsung HD501LJ als Festplatte durchgeführt. Als Betriebssystem wurde Linux (`Kernel 2.6.38 x86_64`) verwendet. Bei allen Tests wurden mindestens 25 unabhängige Läufe durchgeführt und die im Folgenden gezeigten Messwerte als Mittelwerte dieser berechnet. Abhängig vom durchgeführten Test bestand jeder Lauf aus einer vorab festgelegten Menge an Operationen. Tabelle 6.3 listet die entsprechenden Anzahlen auf. Um

Test	Anzahl Operationen pro Lauf
BuildIndex	1
BBoxQuery	50
LocationQuery	50
NearestNeighbor	50
ReadWrite	10.000

TABELLE 6.3: Anzahl Operationen pro Testlauf.

die unterschiedlichen Charakteristika von statischen und dynamischen Geodaten besser erfassen zu können, wurden ferner der kd-tree und der four-dimensional kd-tree in zwei unterschiedlichen Varianten getestet, einmal mit entsprechendem Algorithmus um die jeweilige Gesamtdatenmenge in einer einzigen Operation einzufügen (statisch) und einmal mit einer sequenziellen Vorgehensweise, bei der jedes Feature einzeln eingefügt wurde (dynamisch).

6.2.1 Punktgeometrien als Datenquelle

Als erstes soll das Verhalten der hier diskutieren Indices bei Punktgeometrien vorgestellt werden. Dazu wurden der Layer „Points“ aus dem Datensatz „NRW“ verwendet. Ausschlaggebend hierfür war seine Größe im Vergleich zum anderen Punktgeometrie-Layer „Orte“. Nicht berücksichtigt wurde der four-dimensional kd-tree, da dieser, wie bereits erläutert, nicht für Punktgeometrien geeignet ist.

6.2.1.1 Indexerstellung

Im Rahmen dieses Tests werden, wie bereits im Kapitel 2.4 erläutert, alle Geometrien auf einmal in den Index eingefügt. Es handelt sich also um einen Test, der eher für statische Geodaten von Bedeutung ist. Allerdings ermöglicht dies auch, einen Blick auf die grundsätzlichen Beschaffenheiten der resultierenden Bäume zu werfen, was wiederum auch für andere Szenarien von Interesse ist. Zuerst soll auf die unterschiedlichen Höhen der verschiedenen Indices eingegangen werden, wie sie in Abbildung 6.1 zu sehen sind. Auffällig ist, dass sich drei Gruppen bilden, die jeweils eine ähnliche Höhe aufweisen: die Gruppe mit < 20, $20 - 30$ und > 30. Interessant ist dabei vor allem, dass der bucket pr-quadtree Bestandteil der zweiten Gruppe ist, während alle anderen Indices mit Unterstützung für Buckets in der ersten Gruppe liegen. Hier ist jedoch anzumerken, dass der mx-cif quadtree durch seine Höhenbegrenzung, die für die Testumgebung 8 beträgt, bereits konstruktionsbedingt nur in der ersten Gruppe liegen kann. Damit kann man die

Situation auch so betrachten, dass nur diejenigen Indices in der ersten Gruppe liegen, die grundsätzlich ausbalanciert sind.

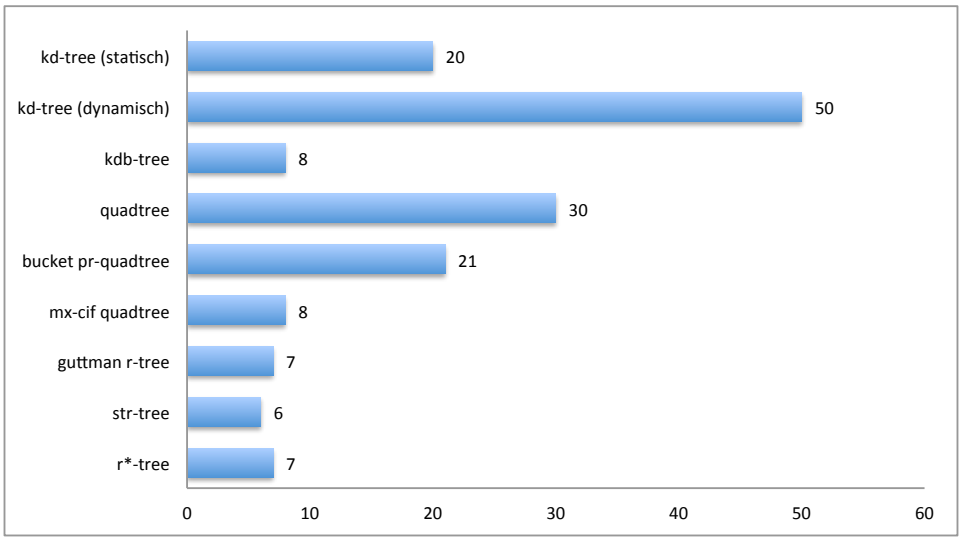

ABBILDUNG 6.1: Höhe der resultierenden Bäume im Test BuildIndex (Datensatz NRW - Punkte)

Ein anderes Bild ist bei den durchschnittlichen Füllgraden[1] aller Knoten in den jeweiligen Bäumen zu sehen (Abbildung 6.4). Hier zeigt vor allem der kdb-tree einen verhältnismäßig schlechten Wert, während der bucket pr-quadtree und mx-cif quadtree sowie der guttman r-tree und r*-tree jeweils fast gleich auf liegen. Gesondert zu betrachten sind die Werte von kd-tree und quadtree, da diese durch das Fehlen von Buckets nicht vergleichbare Charakteristika aufweisen. Ebenso sind die 100% des str-trees konstruktionsbedingt zu erklären, da im ungünstigen Fall gerade mal ein Knoten pro Ebene nicht vollständig gefüllt ist. Bei einer entsprechend hohen Anzahl Knoten ergibt sich so schon durch das Runden ein entsprechender Wert.

Die Gesamtanzahl der Knoten ist in Abbildun 6.3 zu sehen. Auffällig ist in dieser Grafik auf den ersten Blick die sehr hohen und zudem identischen Werte von kd-tree und quadtree. Aber auch hier sei noch einmal an die jeweilige Konstruktionsvorschrift erinnert, durch die sich genau so viele Knoten im Baum ergeben wie Features in der Datenmenge vorhanden sind. Ebenso interessant ist der niedrige Wert des mx-cif quadtrees, doch auch hier spielt die Konstruktionsweise eine entscheidende Rolle, gerade im Hinblick auf die Verwendung von Punktgeometrien. Hieraus ergibt sich auch, dass trotz des sehr geringen Abstands zum str-tree ein doch beachtlicher Unterschied beim Füllgrad zustande kommt. So sei darauf hingewiesen, dass der mx-cif quadtree jeweils nur maximal vier

[1] Der Füllgrad bezeichnet den Anteil der belegten Referenzen (Knoten/Features) in einem Knoten. Leere Referenzen auf Swappages oder innerhalb von Swappages zählen mit.

Kindknoten[2] besitzen kann. Somit ist die Anzahl der inneren Knoten im Verhältnis zu den Blattknoten niedriger als beim str-tree. Gleichzeitig sind diese Referenzen jedoch in jedem Blattknoten leer, d.h. sie wirken sich negativ auf den Füllgrad aus.

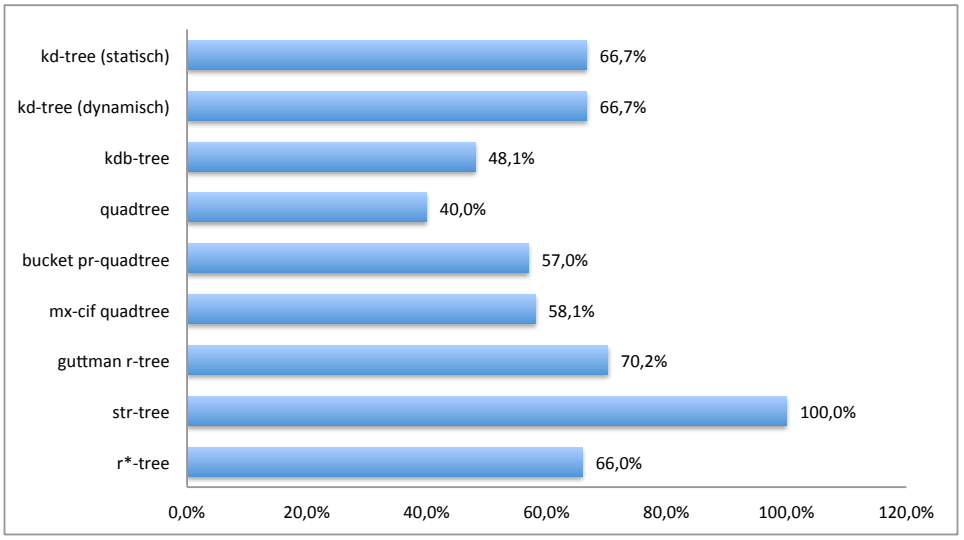

Abbildung 6.2: Füllgrad der Knoten des resultierenden Baums im Test BuildIndex (Datensatz NRW - Punkte)

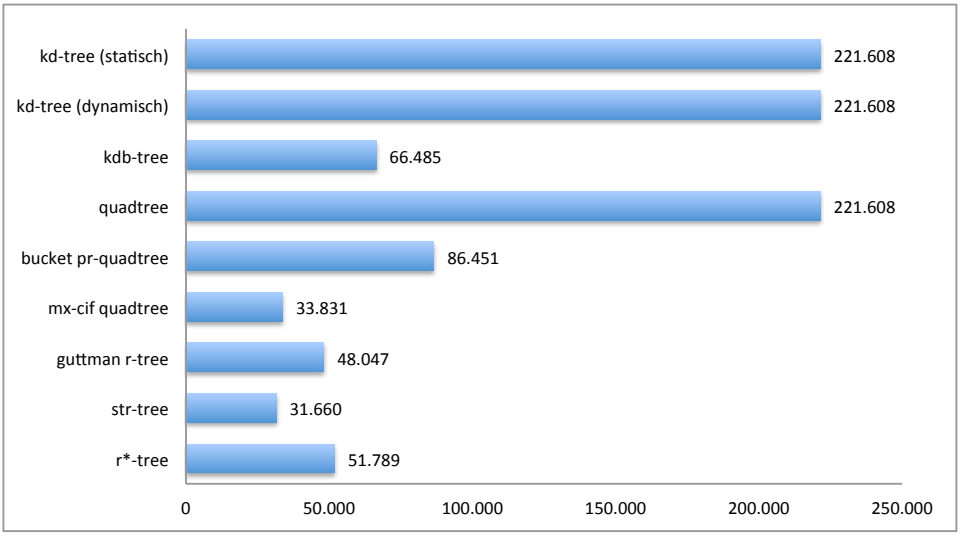

Abbildung 6.3: Gesamtanzahl der Knoten des resultierenden Baums im Test BuildIndex (Datensatz NRW - Punkte)

Ebenso interessant ist die Tatsache, dass erneut der guttman r-tree und der r*-tree sehr dicht bei einander liegen, jedoch wieder ersterer etwas bessere Werte aufweist. Nicht optimal hingegen sind die Daten von kdb-tree und bucket pr-quadtree. An dieser Stelle kann also nun die Vermutung geäußert werden, dass insbesondere die r-trees ein relativ

[2] In der vorliegenden Implementierung wird noch eine weitere Referenz für die Realisierung der Swappages benötigt.

günstiges Laufzeitverhalten zeigen müssten. Als eher problematisch hingegen wäre mit den bisher vorliegenden Daten der bucket pr-quadtree und der kdb-tree einzuschätzen.

Nach dieser ersten Betrachtung empfiehlt es sich nun, die weiteren Daten des Tests anzuschauen, die primär für statische Daten von Interesse sind. Von Bedeutung ist dabei primär die Abbildung 6.4, welche die jeweils durchgeführten Operationen auf den Pages (Knoten) beim Erstellen des Baums zeigt. Vor allem bei den Lese- und Schreiboperationen sei darauf hingewiesen, dass hinter jeder solchen logischen Operation auch eine echte I/O-Operation steht.

Bei der Betrachtung der Abbildung ist gut zu erkennen, dass sich die speziellen Algorithmen zum Einfügen großer Datenmengen (str-tree, kd-tree) deutlich bemerkbar machen. Gleichzeitig sind bei den anderen Indices teilweise erschreckend hohe Lese- und/oder Schreibwerte zu beobachten, wobei hier besonders der r*-tree als Negativbeispiel genannt werden muss. Hierzu korrespondiert auch die gemessene Laufzeit, zu sehen in Abbildung 6.5.

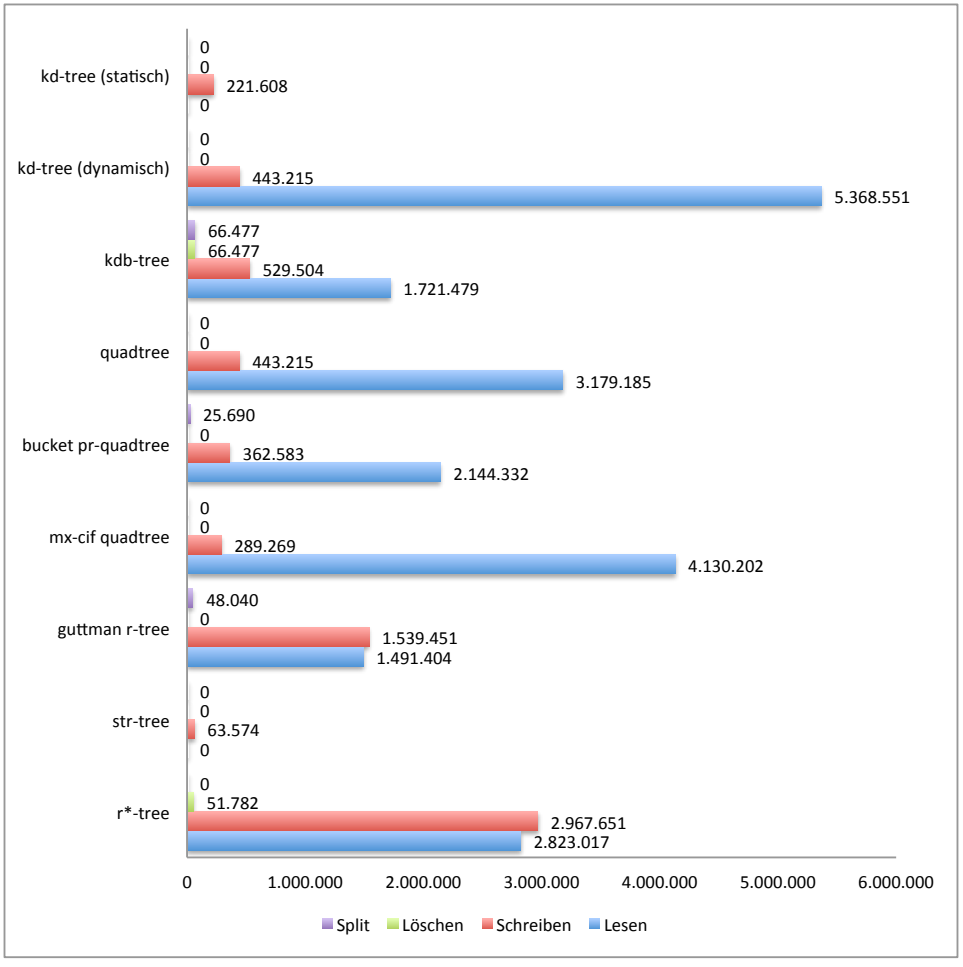

Abbildung 6.4: Anzahl der Operationen auf einzelnen Knoten (Pages) im Test Build-Index (Datensatz NRW - Punkte)

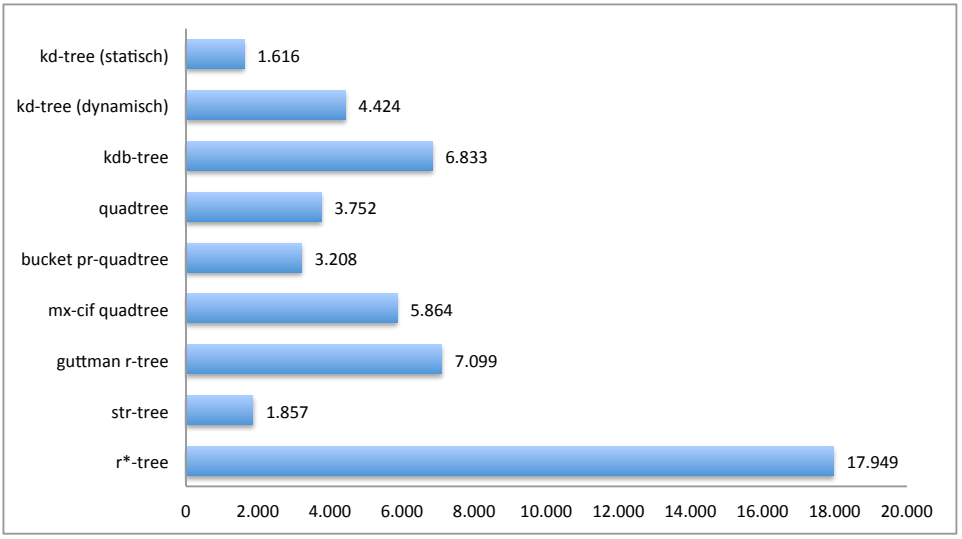

ABBILDUNG 6.5: Gesamtdauer der Einfügeoperationen im Test BuildIndex in Millisekunden (Datensatz NRW - Punkte)

Insgesamt kann geschlussfolgert werden, dass bei der einer regelmäßigen Indizierung sonst statischer Punktdaten ein entsprechender Index mit geeigneter Einfügeroutine gewählt werden sollte, wobei basierend auf dem bisherigen Datenmaterial im wesentlichen der str-tree von Interesse sein dürfte. Eindeutig nicht zu empfehlen hingegen ist der r*-tree, welcher eine relativ hohe Laufzeit und I/O-Last produzieren.

6.2.1.2 Window Queries

Nach der Besprechung der Indexerstellung lohnt es sich nun, einen Blick auf die Suchoperationen im Index zu werfen. Zu Beginn bietet sich die Suche mittels BBox (Window Query) an. Anschließend folgt die Suche des nächsten Nachbarn. Auf eine Analyse der Suche mittels Position wird verzichtet, da dies bei Punktgeometrien nur eingeschränkten Nutzen hat (enthalten sein-Prüfung).

Als erstes soll ein Blick in die Abbildung 6.6 geworfen werden. Hier wird die Gesamtlaufzeit der Suchabfragen gezeigt. Dazu ist als Ergänzung die Abbildung 6.7 hilfreich, die die Anzahl der zu eben diesen Suchanfragen korrespondierenden Lesezugriffe (Knoten/Pages) darstellt. Gut zu erkennen ist, dass die r-trees jeweils eine relativ günstiges Laufzeitverhalten aufzeigen. Deutlich schlechter ist trotz einer vergleichbaren Anzahl an Leseoperationen der mx-cif quadtree. Hier kann vermutet werden, dass die besonderen Bedingungen bei Punktgeometrien die Ursache sind und relativ viele Features jeweils gegen die Suchgeometrie geprüft werden müssen. Ebenfalls von Interessant ist das schlechte abschneiden des bucket pr-quadtrees, welcher sogar langsamer ist als der klassische

quadtree, was so sicherlich nicht zu erwarten war. Eine mögliche Erklärung wäre die Unterteilung des Raums im Kontrast zur Unterteilung der Objektmenge, wodurch sich der quadtree besser an die Gegebenheiten der Objektmenge anpassen kann.

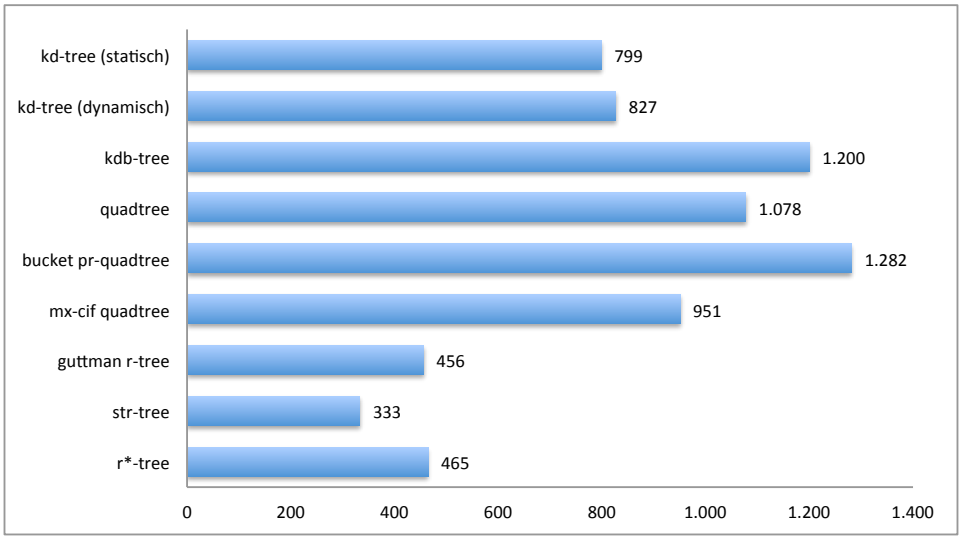

ABBILDUNG 6.6: Gesamtdauer der Suchoperationen im Test BBoxQuery (Datensatz NRW - Punkte) in Millisekunden

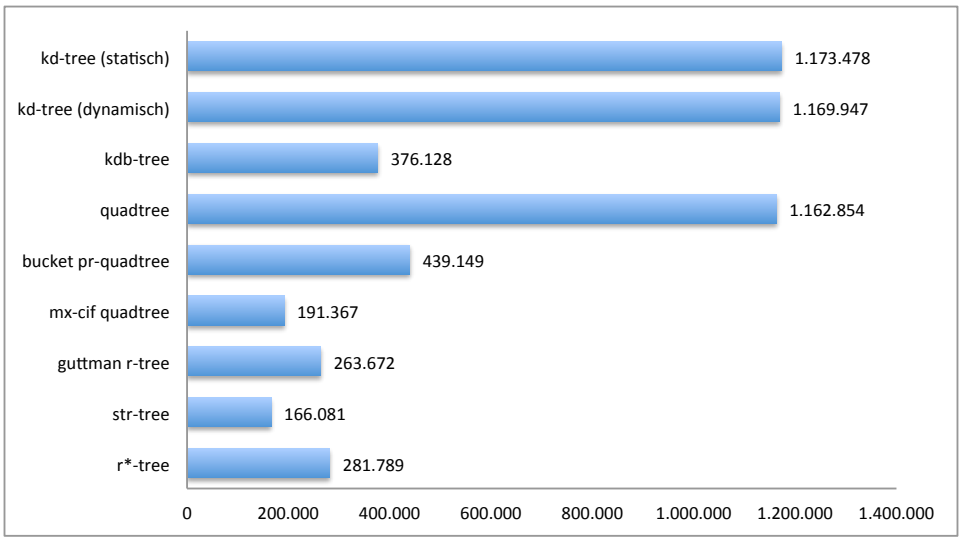

ABBILDUNG 6.7: Anzahl der Lesezugriffe (Pages) im Test BBoxQuery (Datensatz NRW - Punkte)

Ebenso hervorstechend ist die (zu erwartende) hohe Anzahl an Lesezugriffen von kd-tree und quadtree. Was jedoch in dieser Form nicht direkt zu erwarten war ist, dass sich diese nicht im gleichen Maße auf die Laufzeit niederschlagen. Offenbar besitzt auch die Größe der Seiten eine relativ hohe Bedeutung. Abbildung 6.8 zeigt daher im Kontrast zu 6.7 die Anzahl der gelesenen Kilobytes. Hierbei ist gut zu erkennen, dass sich die Verhältnisse

nahezu umgekehrt haben. Gerade zu erschreckend stellen sich nun die Zahlen des kdb-trees dar, was vielleicht auch eine Erklärung für die relativ schlechte Laufzeit darstellt.

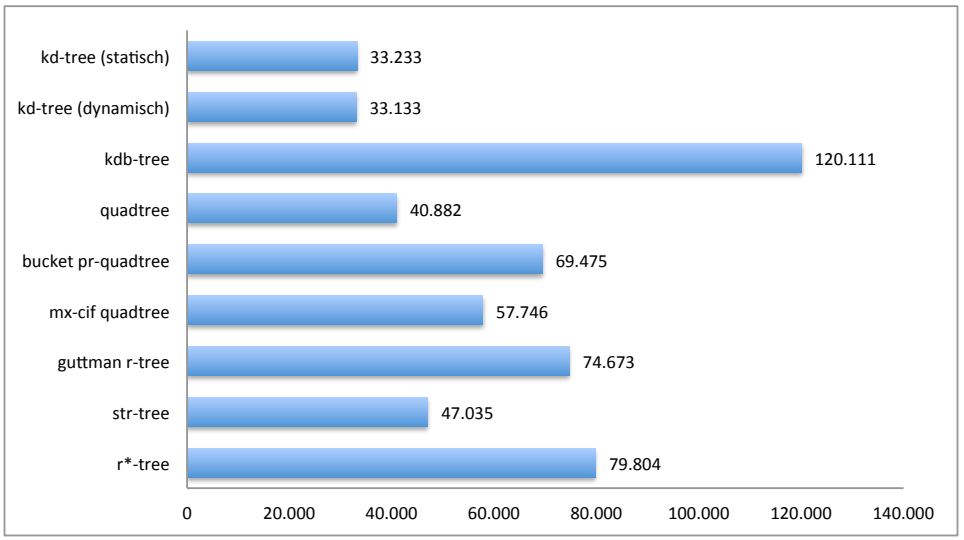

Abbildung 6.8: Anzahl gelesene kB im Test BBoxQuery (Datensatz NRW - Punkte)

6.2.1.3 Nearest Neighbor Queries

Als nächstes soll die zweiten Suchoperation betrachtet werden, die Ermittlung des nächsten Nachbarn. In Abbildung 6.9 sind die Laufzeiten für die einzelnen Indices dargestellt. Auffällig ist, dass konträr zur Window Query hier die quadtrees gegenüber den kd-trees im Vorteil liegen. Vor allem der bucket pr-quadtree zeigt ein auffallend gutes Laufzeitverhalten, wobei er gleichauf mit dem str-tree liegt. Unterstützt wird dieses durch die Abbildung 6.10, die die Anzahl der Leseoperationen zeigt. Überraschend ist in dieser Grafik allerdings, dass der guttman r-tree sowie der r*-tree trotz ihrer relativ guten Laufzeit eine sehr große Menge an Leseoperationen durchführen, welche unter Berücksichtigung der Pagegröße eher kritisch zu bewerten sind.

Ebenfalls aufschlussreich ist das Verhalten des kdb-trees. Wie bereits erwähnt sind bei Nearest Neighbor-Operationen offenkundig quadtrees im Vorteil. Dennoch ist das relativ schlechte Verhalten des kdb-trees überraschend, da dieser sogar gegenüber den klassischen quadtree im Nachteil ist. Es steht zu vermuten, dass hier mehrere Komponenten zusammen kommen, da auch im Rahmen des Window Query-Tests kein gutes Laufzeitverhalten festgestellt werden konnte.

Grundsätzlich kann zum unterschiedlichen Verhalten von quadtrees und kd-trees vermutet werden, dass die unterschiedliche Formung der zu den Knoten korrespondierenden

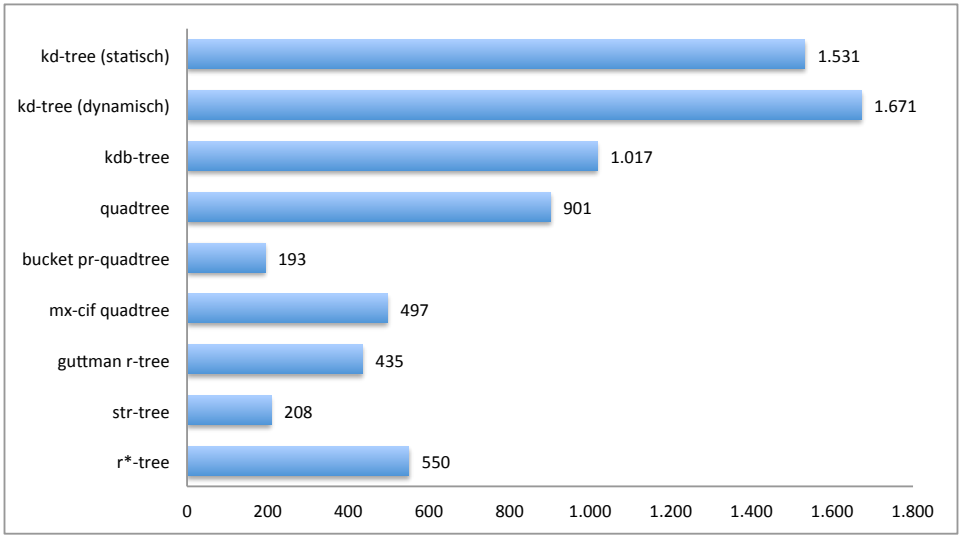

ABBILDUNG 6.9: Gesamtdauer der Suchoperationen im Test NearestNeighbor (Datensatz NRW - Punkte)

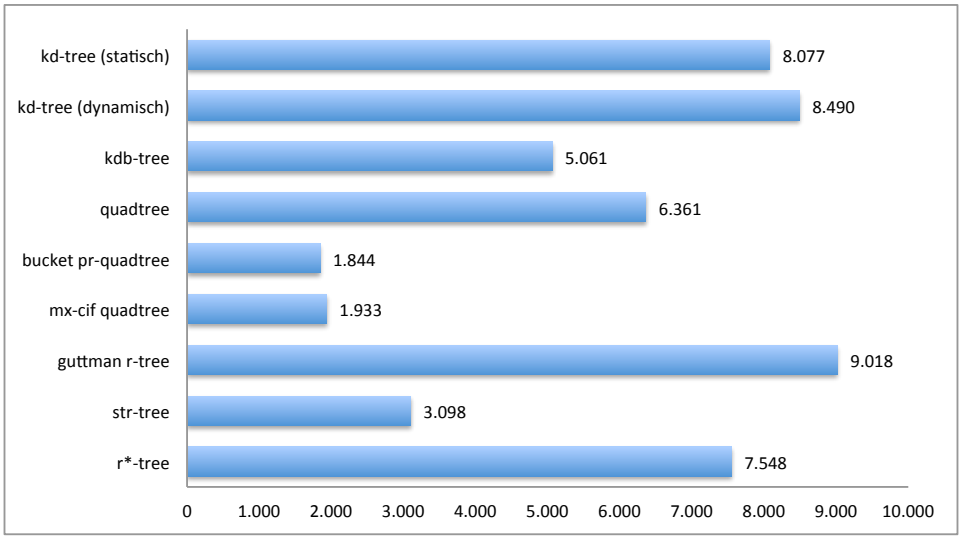

ABBILDUNG 6.10: Anzahl der Lesezugriffe (Pages) im Test NearestNeighbor (Datensatz NRW - Punkte)

„Geometrien“, konkret der Unterschied zwischen Rechteck und Quadrat von entscheidender Bedeutung ist. Die Feststellung, welche Relevanz ein Knoten bei der Ermittlung des nächsten Nachbarn hat, wird über die Distanz bestimmt. Folglich sind möglichst kompakte Knoten im Vorteil, was sich wiederum durch einen geringen Umfang zeigt. Zur Erinnerung sei an dieser Stelle noch einmal an die Kennzahlen des r*-trees (Kapitel 5.7) hingewiesen.

Der besondere Vorteil der quadtrees liegt nun darin, dass die Knoten frei von Überlappungen sind. Dies dürfte auch das im Vergleich zu den r-trees relativ gute Laufzeitverhalten

erklären, da folglich weniger Knoten analysiert werden müssen. Hier könnte auch eine Ursache in der hohen Anzahl der Leseoperationen der beiden dynamischen r-trees liegen. Der str-tree auf der anderen Seite ist bei Punktgeometrien sehr ähnlich zu den quadtrees was die Geometrie der Knoten betrifft. Insbesondere treten keine Überlappungen auf

6.2.1.4 Einfüge- und Löschoperationen

Zum Abschluss der Behandlung von Punktgeometrien sei nun auf das Verhalten bei zufälligen Lese- und Schreibzugriffen im Rahmen des ReadWrite-Tests eingegangen. Naturgemäß keine Rolle spielen dabei die Indices str-tree und kd-tree (statisch). Als erstes sei wie üblich auf das Laufzeitverhalten verwiesen, zu sehen in Abbildung 6.11.

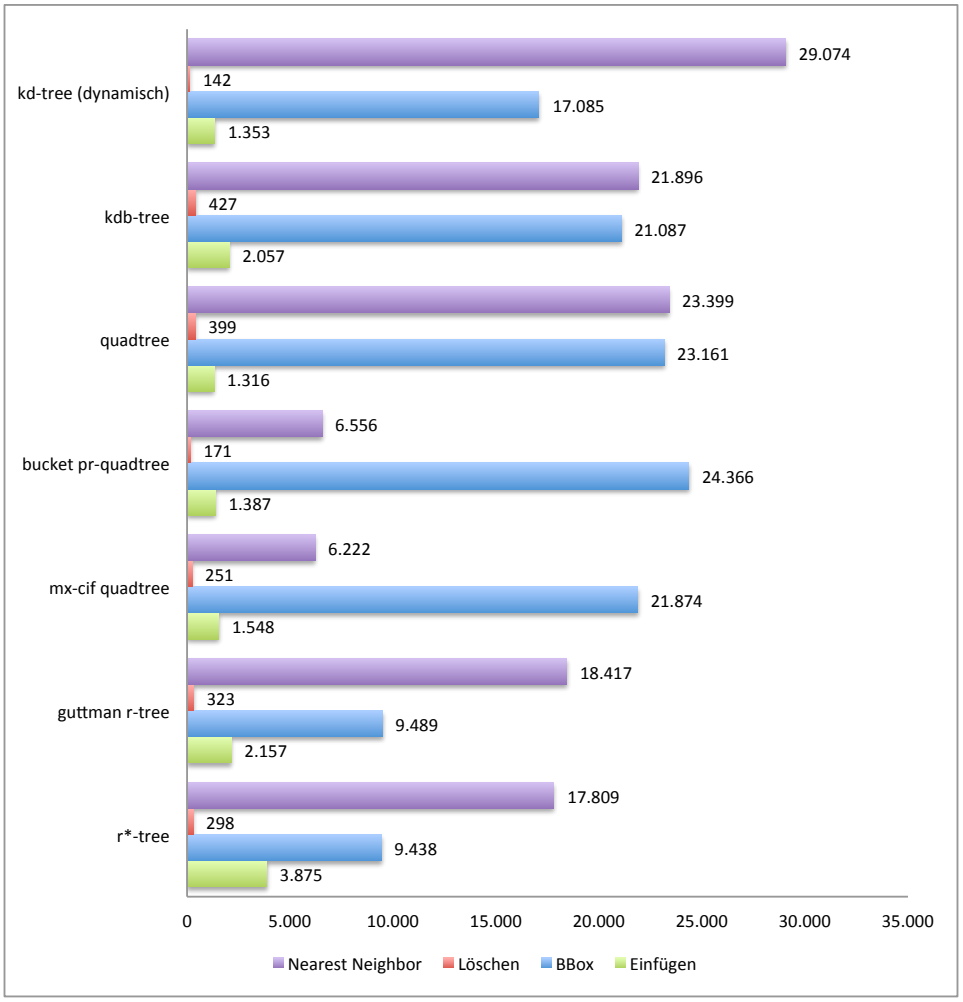

Abbildung 6.11: Laufzeiten der verschiedenen Operationen im ReadWrite-Tests (Datensatz NRW - Punkte)

Gut zu erkennen ist, dass sich das Muster der vorherigen Tests wiederholt. So zeigen für Window Queries die r-trees das beste Verhalten, gefolgt von den kd-trees. Umgekehrt

sieht dies bei Nearest Neighbor-Operationen aus, wo die quadtrees (bucket pr-quadtree und mx-cif quadtree) ein günstigeres Laufzeitverhalten zeigen als die r-trees. Interessant ist, dass dieses Mal der mx-cif quadtrees die geringste Zeit benötigt. Als Ursache kann hier ein besseres Verhalten in Folge der geringeren Objektanzahl bei gleicher Maximalhöhe vermutet werden.

Von Interesse ist das Laufzeitverhalten der Einfüge- und Löschoperationen. Nicht unerwartet ist dieses beim guttman r-tree und r*-tree eher schlecht, wobei auch der kdb-tree ein relativ ungünstiges Verhalten zeigt. Sehr schnell hingegen sind die quadtrees sowie der kd-tree. Einzig bei Löschoperationen weicht hier der klassische quadtree ab. Bei allem ist jedoch zu bedenken, dass Einfüge- und Löschoperationen in der Praxis seltener als entsprechende Suchoperationen vorkommen dürften.

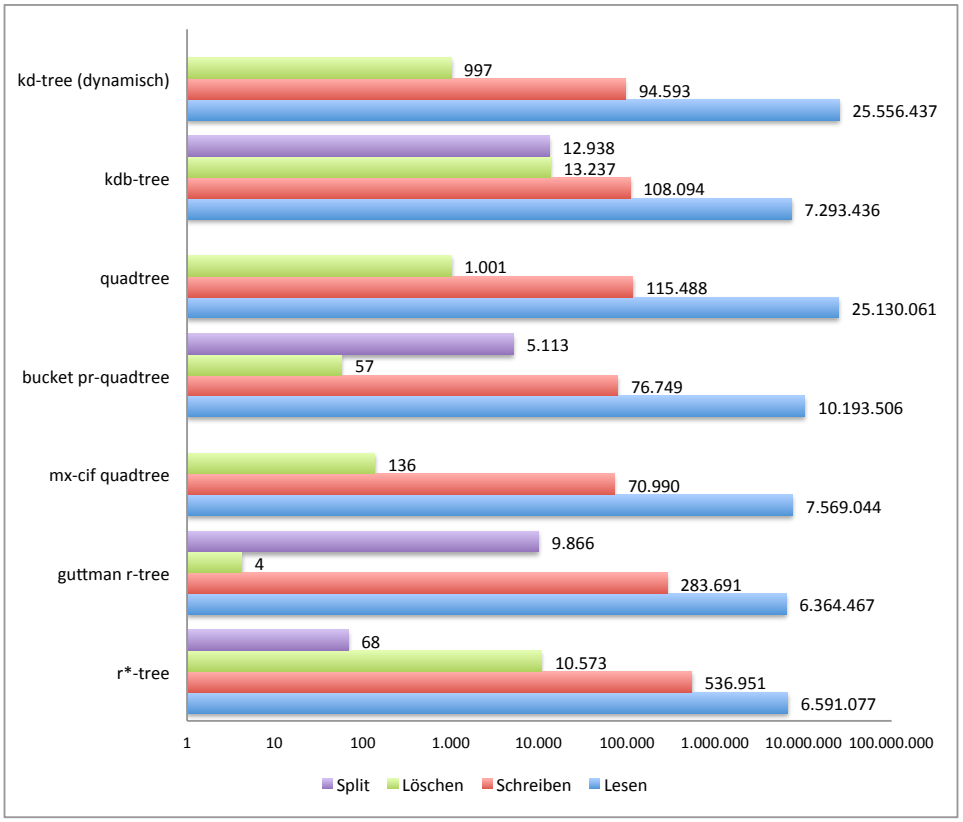

Abbildung 6.12: Anzahl der Operationen auf Knoten/Pages im ReadWrite-Tests (Datensatz NRW - Punkte)

Ergänzend sei dazu ein Blick in die Abbildung 6.12 empfohlen, die die Anzahl der einzelnen Operationen auf den Pages/Knoten auflistet. Hier ist ebenfalls zu beachten, dass eine logarithmische Skala Verwendung findet. Gut zu erkennen ist, dass betreffend der Schreiboperationen sowohl der mx-cif quadtree wie auch der bucket pr-quadtree relativ gute Werte aufweisen. Auch kd-tree und quadtree sind entsprechend gut zu bewerten, da natürlich die Pagegröße berücksichtigt werden muss. Am Rande sei darauf hingewiesen,

dass in dieser Abbildung auch sehr schön das unterschiedliche Verhalten von guttman r-tree und r*-tree betrachtet werden kann bezüglich der Behandlung zu voller Knoten. Während der guttman r-tree einen sehr hohen Wert an Splits aufweist, ist beim r*-tree eine entsprechend große Anzahl an Löschoperationen zu sehen.

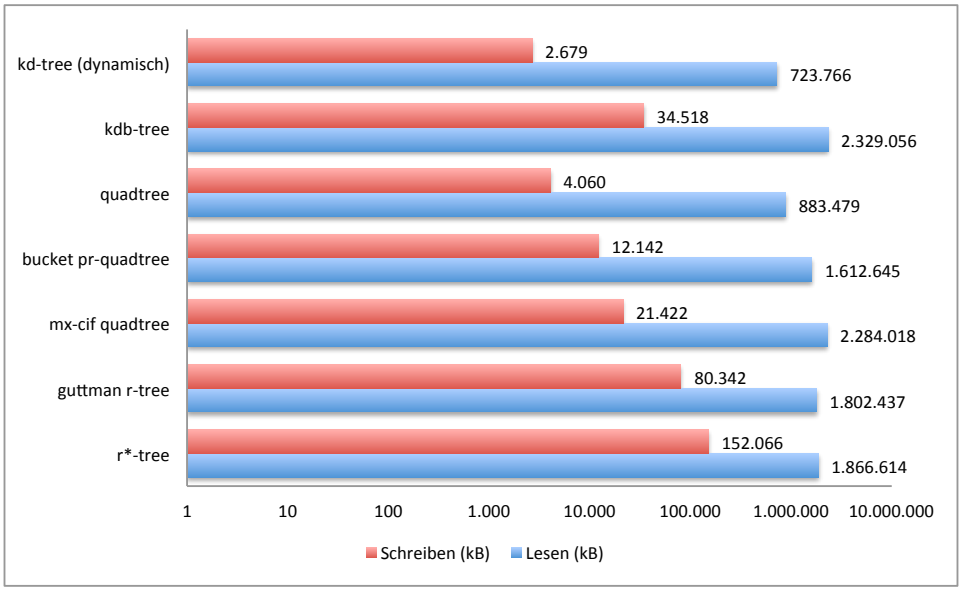

ABBILDUNG 6.13: Gelesene/geschriebene Kilobytes im ReadWrite-Tests (Datensatz NRW - Punkte)

Betreffend der Leseoperationen fällt wiederum auf, dass beide r-trees scheinbar gute Werte erzielen. Dies betrifft aber nur die reine Anzahl. Berücksichtigt man ferner die Pagegröße, so ist kein signifikanter unterschied zu den anderen Indices mit Buckets zu sehen (Abbildung 6.13).

Kurz verwiesen sei noch auf den Status der Bäume nach Abschluss aller Operationen. Abbildung 6.15 zeigt dazu den Füllgrad der Knoten. Vergleicht man diesen mit dem korrespondierenden aus dem BuildIndex-Test, so fällt auf, dass nur der mx-cif quadtree eine größere Abweichung zeigt. Diese kann jedoch auf die feste Maximalhöhe in Kombination mit Punktgeometrien zurückgeführt werden. Einen Kontrast hierzu bietet Abbildung 6.1, die die Höhe der Bäume zeigt. Während die grundsätzlich ausbalancierten Varianten kaum interessantes zeigen (abgesehen von dem minimalen Höhenunterschied zwischen r*-tree und guttman r-tree), zeigen die anderen Indices durchaus ein interessantes Verhalten. Hierzu sei wieder auf die Ergebnisse des BuildIndex-Tests verwiesen. Auffällig ist, dass die maximale Höhe von kd-tree, quadtree und bucket pr-quadtree zwar niedriger als bei Anlage eines vollständigen Index ist, die Werte aber insbesondere beim quadtree nur gering abweichen. Bedenkt man nun, dass nur gut 20% der Features tatsächlich eingefügt sind, so ist dies durchaus bedenklich und kann auf eine gewisse Degenerierung hindeuten.

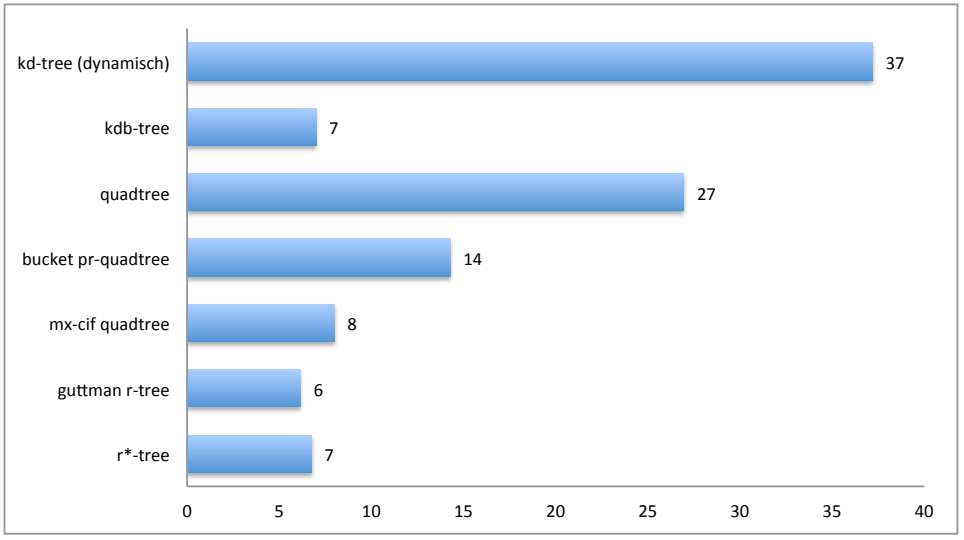

Abbildung 6.14: Höhe des Baums am Ende des ReadWrite-Tests (Datensatz NRW - Punkte)

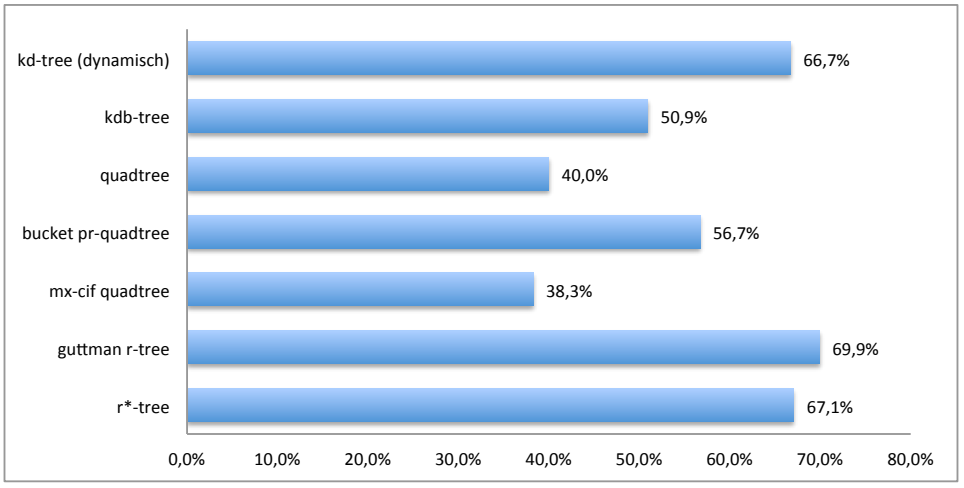

Abbildung 6.15: Füllgrad der Knoten am Ende des ReadWrite-Tests (Datensatz NRW - Punkte)

6.2.1.5 Fazit

Basierend auf den Messergebnissen kann nun für die Indizierung von Punktgeometrien ein entsprechendes Fazit gezogen werden. Solange es sich um statische Daten handelt, so hat sich gezeigt, dass von den betrachteten Indices der str-tree offenbar am besten geeignet ist. Hinsichtlich der Laufzeit konnte immer ein sehr guter Wert beobachtet werden. Auch betreffend der I/O-Operationen ist entsprechendes zu erkennen.

Bezüglich der Indizierung von dynamischen Daten ist dies nicht ganz so eindeutig. Sofern unterschiedliche Suchoperationen benötigt werden, so muss eine Empfehlung zu Gunsten

des guttman r-trees sowie r*-trees ausgesprochen werden, da diese als einzige sowohl bei Window Queries als auch bei Nearest Neighbor-Operationen ein günstiges Laufzeitverhalten gezeigt haben. Kann jedoch auf Window Queries verzichtet werden, so wäre auch die Wahl des bucket pr-quadtrees oder mx-cif quadtrees[3] möglich. Beide zeigen zudem eine recht hohe Geschwindigkeit bei Einfüge- und Löschoperationen, auch bedingt durch die geringe Anzahl an nötigen Schreibzugriffen. Darüber hinaus kann insbesondere der bucket pr-quadtree mit relativ geringem Aufwand implementiert werden. Bei kleineren Datenmengen können natürlich auch der kd-tree oder quadtree Verwendung finden.

Interessant ist das ungünstige Verhalten des kdb-trees. Es kann vermutet werden, dass hier die Erweiterung auf Nicht-Punktgeometrien die Ursache sind. Konkret dürften sich dabei sowohl die gestiegene Pagegröße als auch die komplexeren Algorithmen die Ursache hierfür darstellen.

Als unerwartet hingegen muss die geringere Geschwindigkeit des r*-trees bezeichnet werden. Zwar ist diese nicht signifikant, aber dennoch durchgehend zu beobachten. Erwartet worden wäre hingegen ein signifikant besseres Verhalten als das des guttman r-trees. Möglicherweise sind die gewählten Kennzahlen bei Nutzung von Geodaten und insbesondere Punktgeometrien nicht geeignet. Gleichwohl können natürlich auch Probleme in der Implementierung nicht ausgeschlossen werden.

Zur weiteren Überprüfung der Messwerte gerade auch hinsichtlich der Ergebnisse des r*-trees wurde auch ein Lauf der Testumgebung mit einer Bucket-Größe von 24 durchgeführt. Abgesehen von den zu erwartenden Abweichungen (Höhe der Bäume, CPU-intensivere Berechnung) wurden jedoch nur wenig weiteren Erkenntnisse bezüglich der Problematik festgestellt. Allein der kdb-tree zeigt eine entsprechend positive Tendenz. Um den geneigten Leser jedoch einen Eindruck zu vermitteln sei an dieser Stelle auf Abbildung 6.16 verwiesen, in die Laufzeiten im ReadWrite-Test für die Bucket-basierten Indices dargestellt werden.

6.2.2 Nicht-Punktgeometrien als Datenquelle

Nach der Betrachtung des Verhaltens bei Nutzung von Punktgeometrien soll nun auf das Verhalten bei Verwendung zweiten betrachteten Geometrietyps eingegangen werden. Die Basis stellt dabei der Layer „buildings“ aus dem NRW-Datensatz dar. Untersucht werden die Indices four-dimensional kd-tree[4], kdb-tree, mx-cif quadtree, guttman r-tree,

[3] Die Wahl des mx-cif quadtrees setzt die Ermittlung einer geeigneten Maximalhöhe voraus.
[4] Der four-dimensional kd-tree wird in den Diagrammen als der Übersichtlichkeit halber als 4d kd-tree bezeichnet.

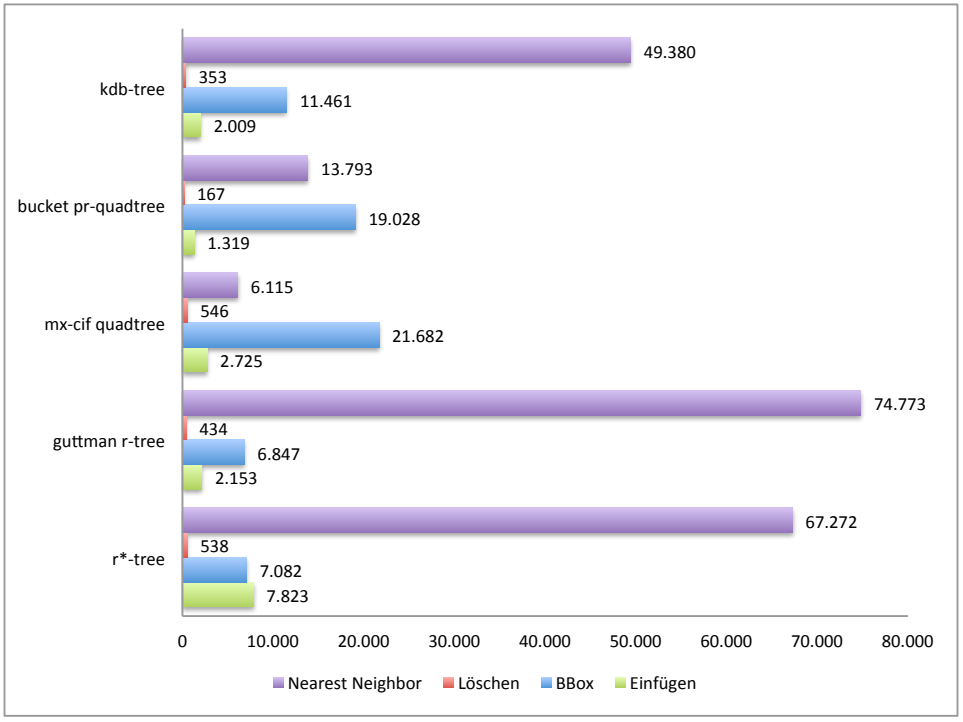

ABBILDUNG 6.16: Laufzeiten der verschiedenen Operationen im ReadWrite-Tests (Datensatz NRW - Punkte - Bucketgröße 24)

str-tree und r*-tree. Die Abfolge der betrachteten Testfälle ist dabei analog zu der bei Betrachtung von Punktgeometrien, jedoch erweitert um den Test von Location Queries.

6.2.2.1 Indexerstellung

Als erstes sei auf das Verhalten bei Erstellung eines Index eingegangen, konkret auf die Eigenschaften der resultierenden Bäume. Dazu sei auf die Abbildung 6.17 verwiesen, die die Höhe der resultierenden Bäume zeigt. Auffällig ist, dass die Höhenunterschiede bei Nicht-Punktgeometrien relativ gering ausfallen. Lediglich der four-dimensional kd-tree weißt insbesondere in der dynamischen Variante einen stark erhöhten Wert auf.

Ebenfalls relativ uniform stellt sich der Füllgrad der Knoten dar, zu sehen in Abbildung 6.19. Hier weichen lediglich der kdb-tree nach unten sowie der str-tree nach oben ab. Letzteres war jedoch bedingt durch die Eigenschaften des Index zu erwarten gewesen. Ergänzend sei dann noch auf die Gesamtanzahl der Knoten verwiesen, die in Abbildung **??** gezeigt wird. Hier bestätigt sich in gewisser Weise das bereits erkennbare Bild. Dabei fällt insbesondere die verhältnismäßig hohe Anzahl an Knoten beim kdb-tree ins Auge. Dies wäre eigentlich bei einem Index mit Bucket-Unterstützung nicht zu erwarten gewesen. Hierdurch ist aber wiederum der geringe Füllgrad zu erklären. Die hohen Werte des four-dimensional kd-trees können als konstruktionsbedingt eingestuft werden. Überraschend

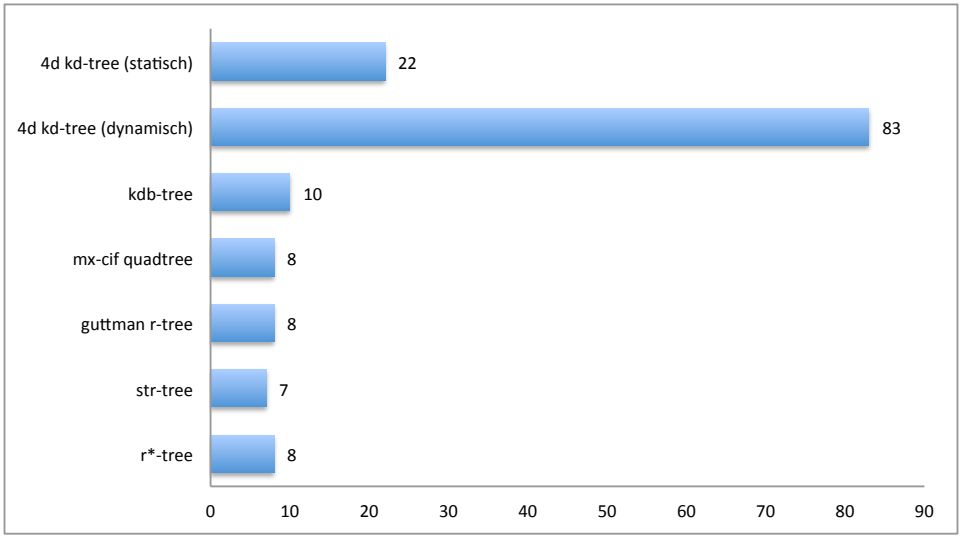

ABBILDUNG 6.17: Höhe der aus dem BuildIndex-Test resultierenden Bäume (Datensatz NRW - Gebäude)

hingegen ist der sehr niedrige Wert des mx-cif quadtrees auch gegenüber dem str-tree. Hier dürfte sich bemerkbar machen, dass auch in inneren Knoten Features referenziert werden können.

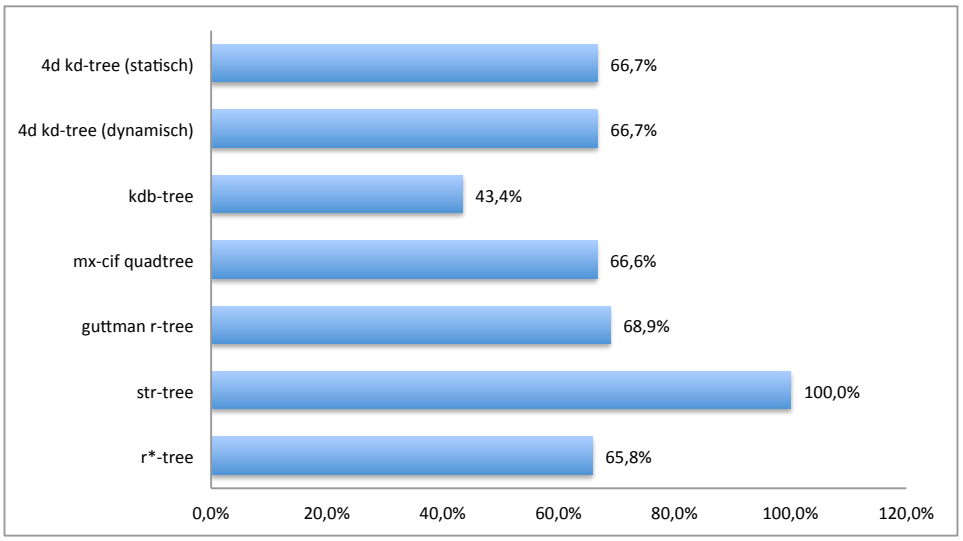

ABBILDUNG 6.18: Füllgrad der aus dem BuildIndex-Test resultierenden Bäume (Datensatz NRW - Gebäude)

Nach Betrachtung der gezeigten Charakteristika kann bis zu diesem Punkt eine erste Bilanz gezogen werden. Ein relativ gutes Verhalten kann auch basierend auf den Erfahrungen bei Punktgeometrien dem str-tree unterstellt werden. Auch die anderen r-trees besitzen gute Ausgangswerte. Als problematisch hingegen kann bereits der kdb-tree eingeschätzt werden, ebenso wie der four-dimensional kd-tree.

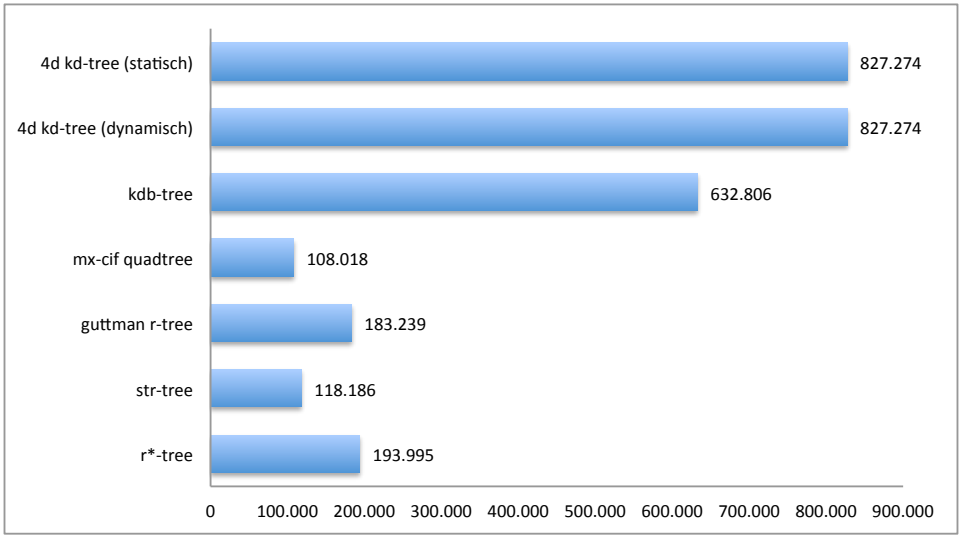

ABBILDUNG 6.19: Gesamtanzahl der Knoten der aus dem BuildIndex-Test resultierenden Bäume (Datensatz NRW - Gebäude)

Nach Betrachtung der allgemeinen Eigenschaften sei nun noch auf die Kosten der Indexerstellung eingegangen, wie sie insbesondere für statische Geodaten von Interesse sind. Dazu sei auf Abbildung 6.20 hingewiesen, in der die Gesamtdauer der Einfügeoperationen in Millisekunden dargestellt wird. Auffallend ist die relativ ähnliche Laufzeit bei kdb-tree, mx-cif quadtree und r*-tree. Während beim four-dimensional kd-tree eine entsprechend höhere Geschwindigkeit durch seine einfache Struktur und beim str-tree durch seinen Algorithmus vermutet werden konnte, ist es umso überraschender, dass auch der guttman r-tree einen relativ guten Wert erzielt.

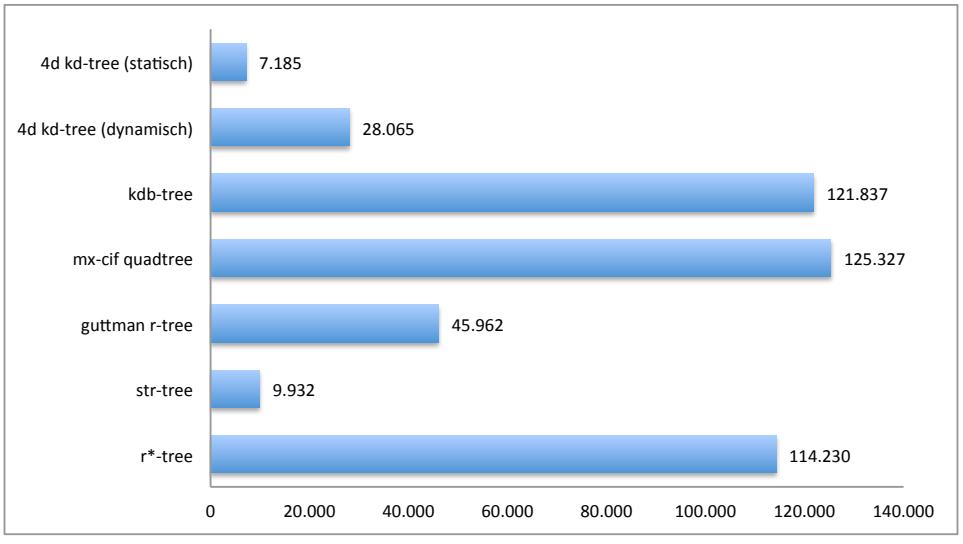

ABBILDUNG 6.20: Gesamtdauer der Einfügeoperationen des BuildIndex-Test in Millisekunden (Datensatz NRW - Gebäude)

Als Kontrast hierzu kann die Abbildung 6.21 gezählt werden. Gerade die relativ gute Laufzeit des guttman r-trees korrespondiert scheinbar nicht mit der Anzahl der konkret durchgeführten Operationen auf den einzelnen Knoten. Dort weisen vielmehr die statischen Indices sehr gute Werte auf - was aber nur im Falle des four-dimensional kd-trees mit der Laufzeit korrespondiert. Gleichwohl zeigt der guttman r-tree im Verhältnis zum r*-tree und mx-cif quadtree geringere Werte.

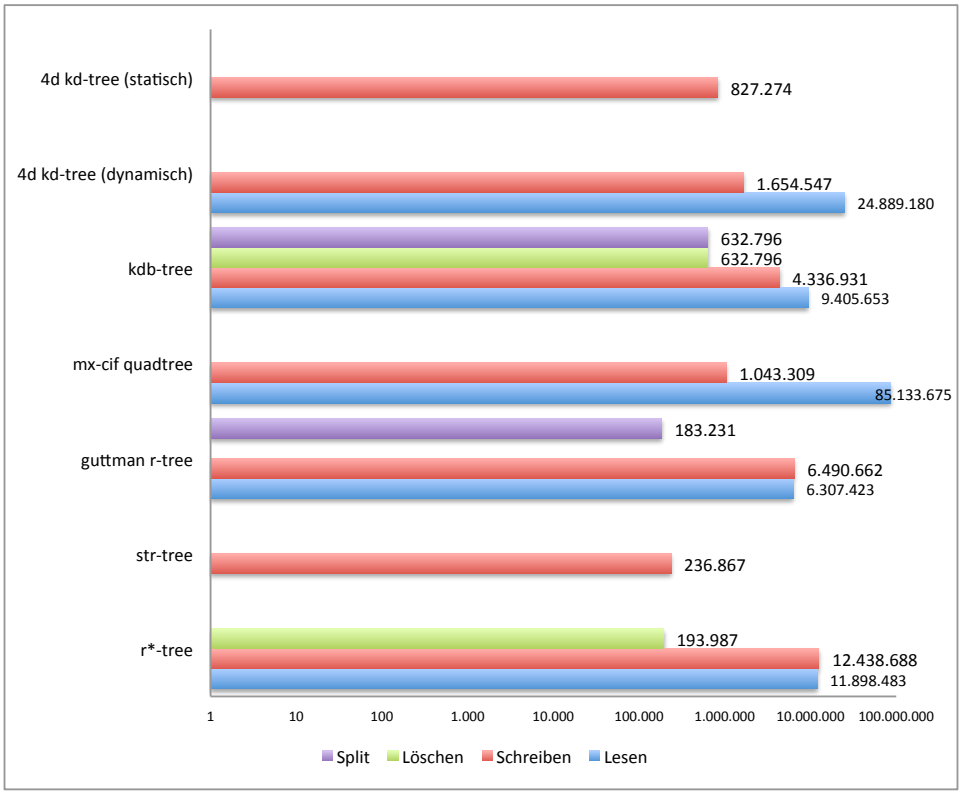

Abbildung 6.21: Anzahl der Operationen auf einzelnen Knoten (Pages) im BuildIndex-Test (Datensatz NRW - Gebäude)

6.2.2.2 Window Queries

Als erste Suchoperation soll nun die Window Query näher analysiert werden. Zum Einstieg ist dazu ein Blick auf die Gesamtlaufzeit empfehlenswert, die in der Grafik **??** gezeigt wird. Es wird schnell deutlich, dass offenbar der kdb-tree ein Laufzeitproblem bei dieser Form der Abfragen hat, da dessen Wert deutlich über allen anderen liegt. Betrachtet man die verbliebenen Indices, so fällt ferner auf, dass der str-tree offenbar das beste Laufzeitverhalten bietet, gefolgt von den anderen r-trees. Interessant ist ferner, dass der mx-cif quadtree einen schlechteren Wert als der four-dimensional kd-tree in beiden Varianten aufweist, wobei zudem hervorzuheben ist, dass die dynamische Variante dieses Index einen besseren Wert als die statische hat.

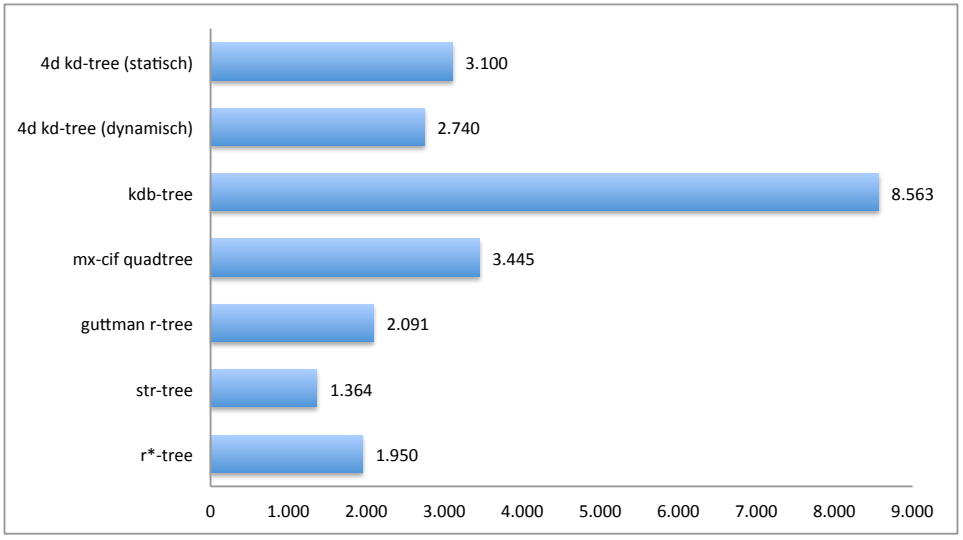

ABBILDUNG 6.22: Gesamtdauer der Suchoperationen im BBoxQuery-Test in Millisekunden (Datensatz NRW - Gebäude)

Zur weiteren Analyse lohnt sich daher ein Blick auf die Anzahl der Lesezugriffe (Abbildung 6.23) sowie der gelesenen Kilobyte (Abbildung 6.24). Insbesondere letztere zeigt das relativ schlechte Verhalten des kdb-trees, welcher fast eine Größenordnung mehr Daten lesen muss als andere Indices. Auf der anderen Seite bestätigen beide Grafiken auch das sehr günstige Verhalten des str-trees. Verwunderlich ist aber, dass auch der mx-cif quadtree hinsichtlich der Leseoperationen scheinbar ein gutes Verhalten besitzt. Der Grund für das schlechte Laufzeitverhalten ist folglich in rechenintensiven Checks der MBRs zu suchen, welche durch die Struktur des Baums auf jeder Ebene durchgeführt werden müssen.

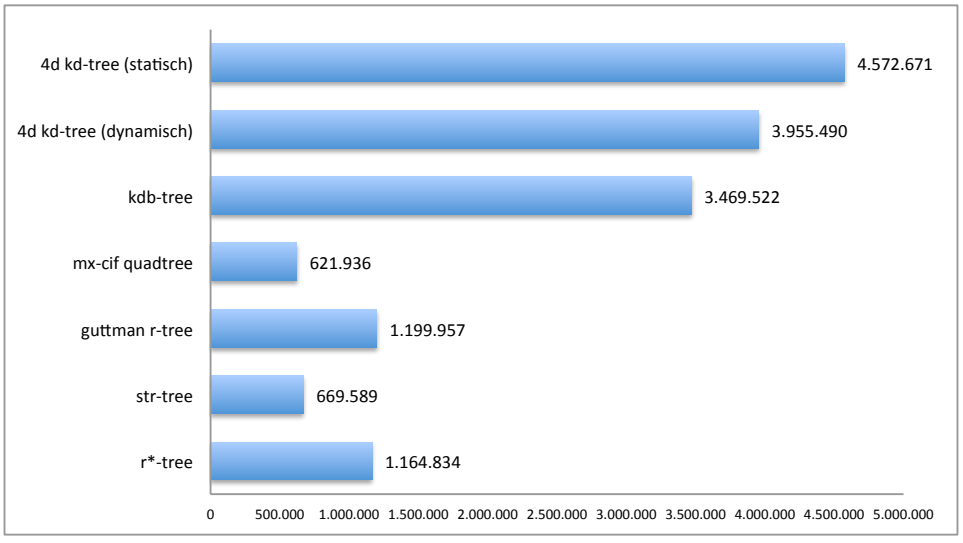

ABBILDUNG 6.23: Anzahl der Lesezugriffe (Pages) im Test BBoxQuery (Datensatz NRW - Gebäude)

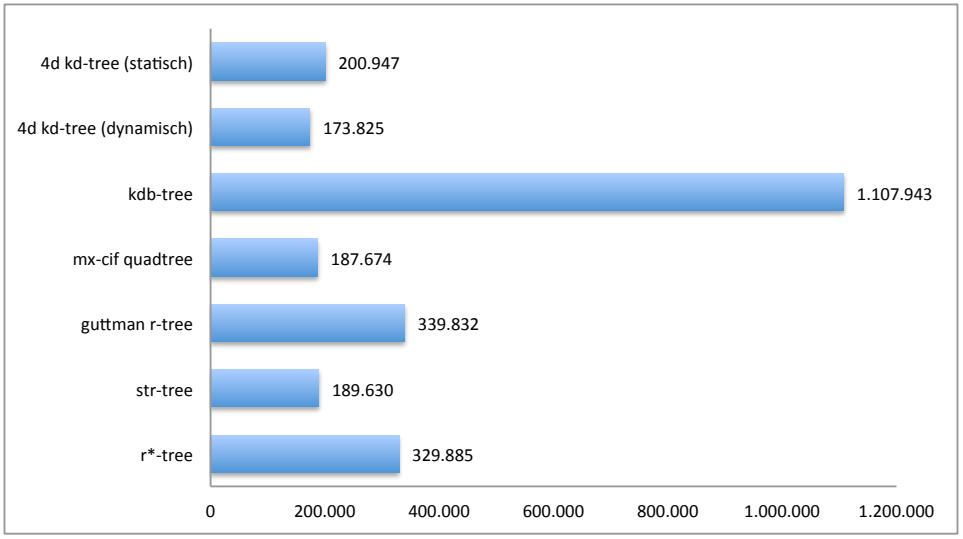

ABBILDUNG 6.24: Anzahl der gelesenen Kilobyte im Test BBoxQuery (Datensatz NRW - Gebäude)

6.2.2.3 Location Queries

Als zweite Suchoperation ist gerade für Nicht-Punktgeometrien die Location Query von Bedeutung. Für die meisten Indices stellt diese jedoch kein Problem dar, wie gut in Abbildung 6.25 zu sehen ist. Allein der four-dimensional kd-tree insbesondere in der statischen Variante zeigt hier ein ungünstiges Verhalten. Alle anderen Indices weisen weitestgehend gleiche Werte auf, auch wenn eine gewisse Differenzierung zwischen den r-trees und kdb-tree sowie mx-cif quadtree festgestellt werden kann. Es muss dabei aber bedacht werden, dass durch die sehr geringen Abstände und generell sehr kleinen Werte dies nur begrenzte Aussagekraft besitzt. Zudem weisen die Werte des mx-cif quadtrees eine sehr hohe Standardabweichung auf.

Gleichwohl zeigt sich ein ähnliches Bild in Abbildung 6.26, welche die Summe der gelesenen Kilobyte pro Index auflistet. Auch hier ist zu erkennen, dass der four-dimensional kd-tree einen schlechten Wert erreicht. Auch die weiteren gemessenen Laufzeitunterschiede spiegeln sich entsprechend im Umfang der Leseoperationen wieder, ausgenommen jedoch des kdb-trees. Dieser zeigt allein bei Betrachtung der Leseoperationen ein sehr gutes Bild.

6.2.2.4 Nearest Neighbor Queries

Die letzte der hier betrachteten Suchoperationen ist die Nearest Neighbor-Query. Gleichzeitig ist bei diesem Test auch der größte Laufzeitunterschied zwischen einzelnen Indices zu beobachten. Wie in Abbildung 6.27 zu sehen ist, zeigt hierbei der four-dimensional kd-tree ein nicht akzeptables Verhalten, in dem dieser Größenordnungen mehr Zeit zum

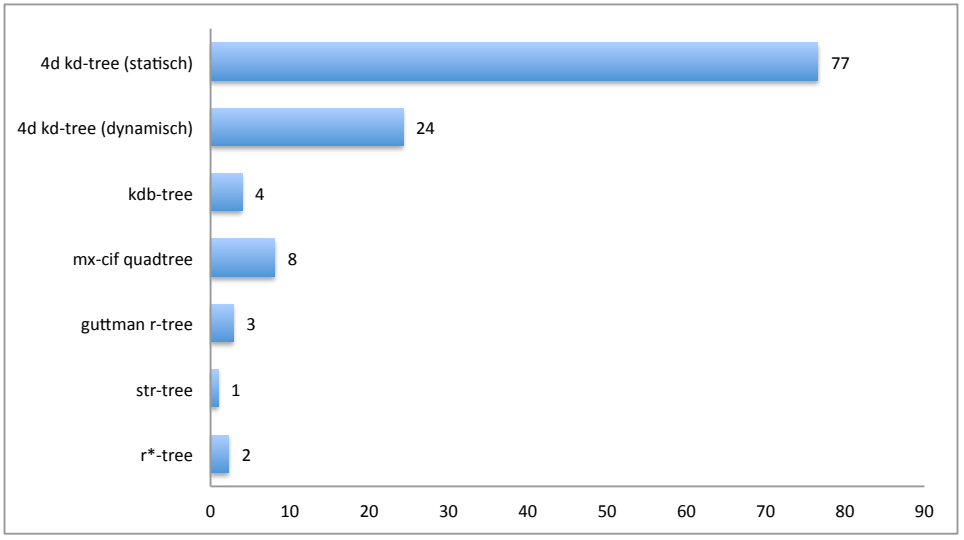

Abbildung 6.25: Gesamtdauer der Suchoperationen im LocationQuery-Test in Millisekunden (Datensatz NRW - Gebäude)

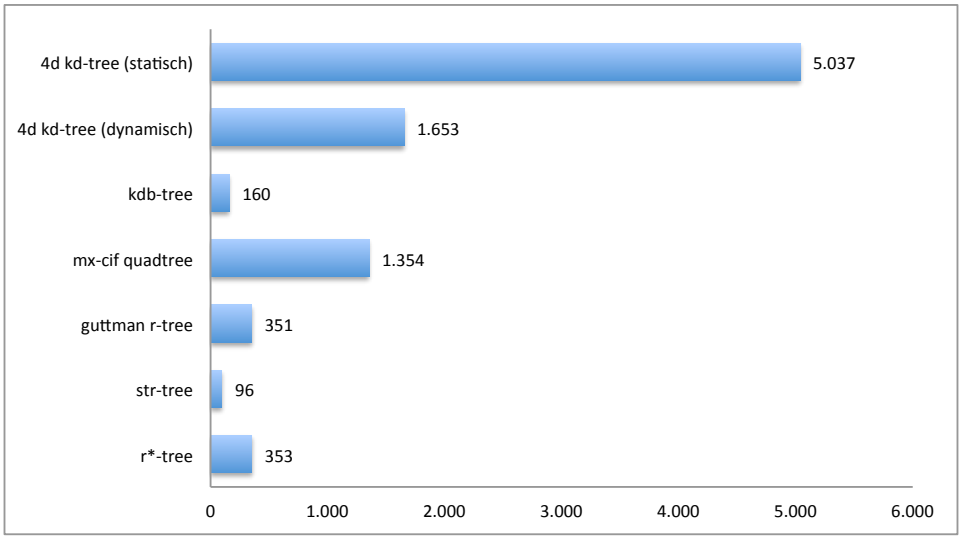

Abbildung 6.26: Anzahl der gelesenen Kilobyte im Test LocationQuery (Datensatz NRW - Gebäude)

Absolvieren der Tests benötigt als alle anderen. Die restlichen Indices zeigen zwar ebenfalls gewisse Unterschiede, jedoch bedeutend geringere.

Von Interesse ist, dass bei Verwendung von Punktgeometrien ein anderes Bild beobachtet werden kann als hier betreffend mx-cif quadtree und kdb-tree. Während bei Punktgeometrien ersterer ein klar besseres Laufzeitverhalten demonstriert ist hier der kdb-tree überlegen. Dies ist gerade vor dem Hintergrund der sonst relativ schlechten Performance bei Nicht-Punktgeometrien überraschend. Auch ist hier bei der Interpretation gewisse

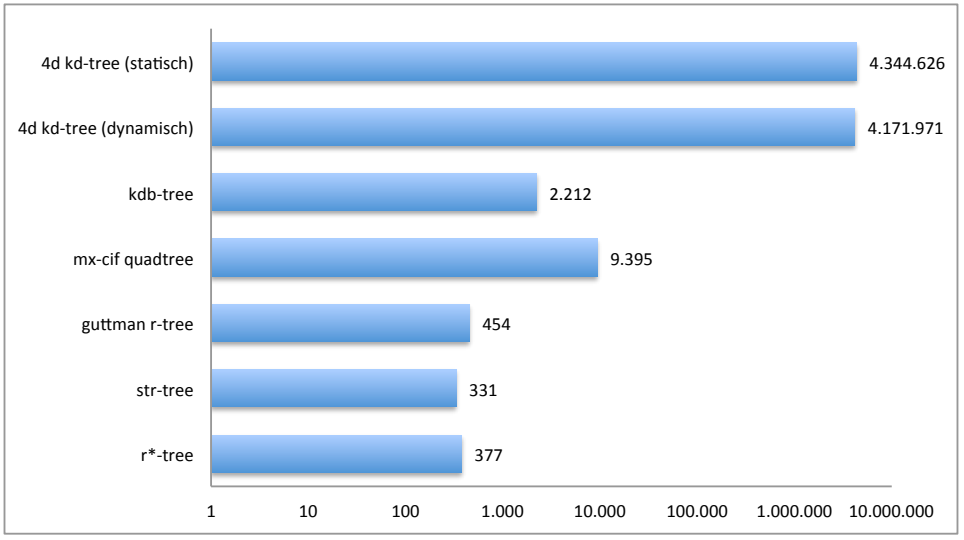

ABBILDUNG 6.27: Gesamtdauer der Suchoperationen im NearestNeighbor-Test in Millisekunden (Datensatz NRW - Gebäude)

Vorsicht geboten, da dieser Index eine Standardabweichung von fast 40% des Mittelwerts aufweist.

In gewissem Umfang setzt sich dieses Bild auch bei Betrachtung der Leseoperationen fort (Abbildung 6.28) wobei natürlich auch wieder die Größe der Knoten/Pages zu berücksichtigen ist (Abbildung 6.29). Zwar lässt sich einwenden, dass der kdb-tree hier dennoch klar hinter dem mx-cif quadtree liegt, doch ist dabei zu bedenken, dass der Unterschied keineswegs so klar ist, wie beispielsweise beim BBoxQuery-Test.

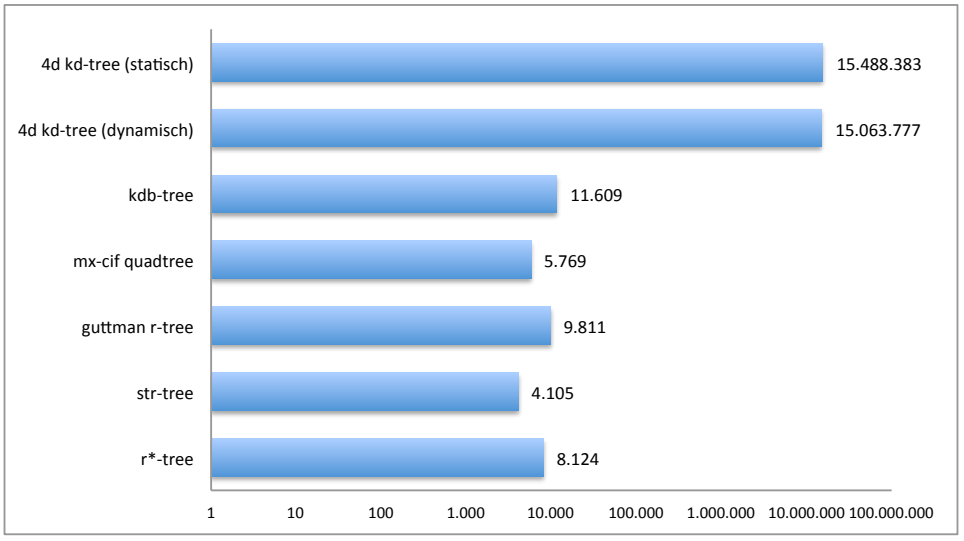

ABBILDUNG 6.28: Anzahl der Lesezugriffe (Pages) im Test Nearest Neighbor (Datensatz NRW - Gebäude)

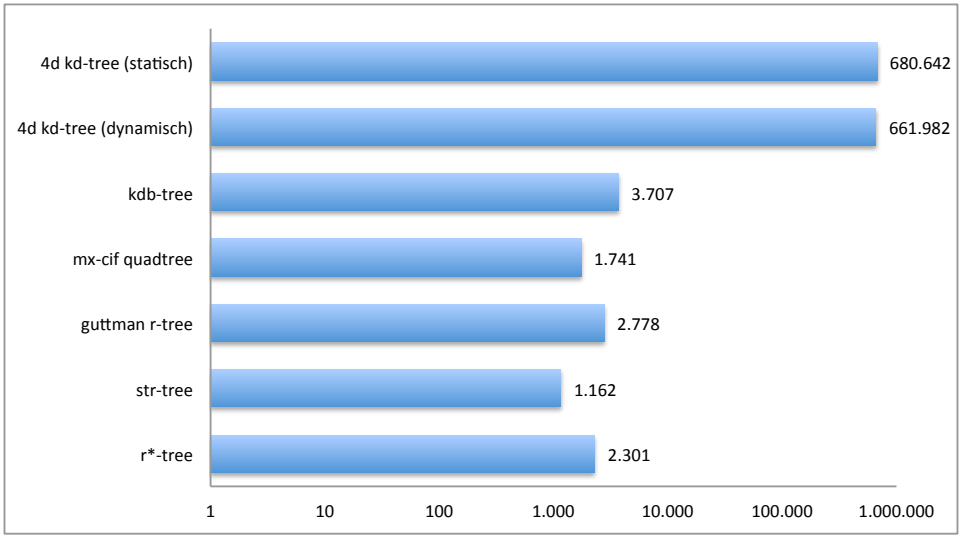

ABBILDUNG 6.29: Anzahl gelesene kB im Test Nearest Neighbor (Datensatz NRW - Gebäude)

6.2.2.5 Einfüge- und Löschoperationen

Zum Abschluss soll nun auf das Verhalten bei dynamischen Daten eingegangen werden. Analog zur Betrachtung der Punktgeometrien können hier naturgemäß nicht alle Indices berücksichtigt werden, da nicht alle für dynamische Daten geeignet sind. Als Einstieg bietet sich wieder ein Blick auf das Laufzeitverhalten an, dargestellt in Abbildung 6.30. Zu beachten ist, dass die Abbildung der Übersichtlichkeit halber eine logarithmische Skala verwendet. Dennoch sticht sofort auf den ersten Blick der four-dimensional kd-tree hervor, um genau zu sein dessen Verhalten bei Nearest Neighbor-Abfragen. Dies ist also vergleichbar zu den bereits beim Einzeltest vorgefundenen Verhalten. Auf der anderen Seite weisen die r-trees relativ günstige Laufzeiteigenschaften auf. Auch dies deckt sich mit den bereits separat vorgefundenen Verhalten.

Auch für die Window-Query kann beobachtet werden, dass die Differenzierung zwischen den einzelnen Indices analog zu der aus dem Einzeltest ist. Somit bleibt nur das Lesen und Schreiben als solches als neuer Aspekt. Bei beiden Operationen fällt dabei auf, dass sowohl four-dimensional kd-tree als auch mx-cif quadtree relativ gute Werte erzielen. Den gegenüber stehen die r-trees, welche die meiste Zeit benötigen. Anzumerken ist noch, dass der kdb-tree hinsichtlich der Einordnung etwas problematisch ist, da zwar die Einfügeoperation sehr viel Zeit benötigt (langsamer als der r*-tree), jedoch auf der anderen Seite die Löschoperation deutlich schneller ist als bei beiden r-trees. Anzumerken ist jedoch, dass die Löschoperationen fast durchweg eine sehr hohe Standardabweichung aufweisen. Insbesondere der kdb-tree fällt hier mit vielen Ausreißern nach oben auf, die eine Standardabweichung von mehr als dem Mittelwert bewirken.

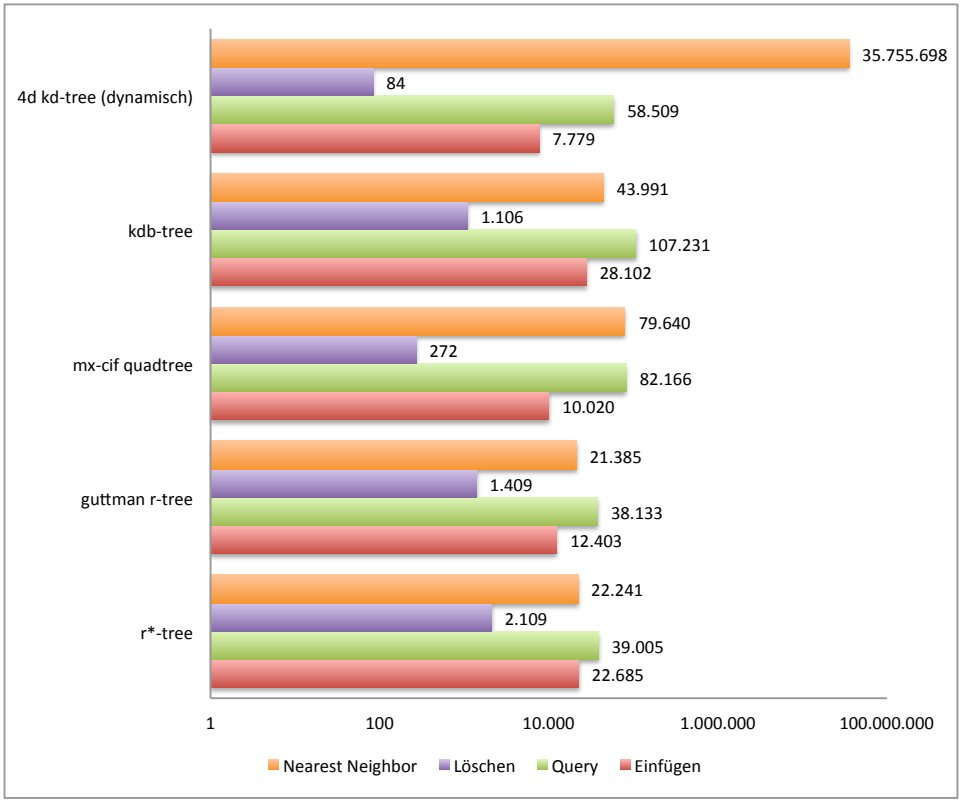

ABBILDUNG 6.30: Laufzeiten der verschiedenen Operationen im ReadWrite-Test (Datensatz NRW - Gebäude)

Betrachtet man nun ergänzend die Anzahl der I/O-Operationen (Abbildung 6.31), so fällt hier ein ähnliches Bild auf wie bei den vorherigen Tests. Wie zu erwarten war, sticht hier ebenfalls der four-dimensional kd-tree hervor, was auf die Nearest Neighbor-Abfragen zurück zu führen sein dürfte. Die besten Werte erzielt wiederum der mx-cif quadtree, was auch schon in den vorhergehenden Tests zu beobachten war. Als problematisch muss wiedereinmal der kdb-tree eingestuft werden, da zu den ohnehin nicht optimalen Werte auch noch die Knoten/Page-Größe hinzu kommt.

Nun lohnt es sich noch einen Blick auf die Eigenschaften der aus den Einzeloperationen resultierenden Bäume zu werfen. Während die Höhe dieser (Abbildung 6.32) noch keine Überraschungen auch im Vergleich zum BuildIndex-Test bereit hält, so fällt beim Vergleich der Füllgrade (Abbildung 6.33) durchaus eine gewisse Abweichung auf. Konkret besitzt der kdb-tree einen leicht höheren Füllgrad, während der mx-cif quadtree nach unten abweicht. Letzteres konnte durch die feste Höhe in Kombination mit der geringeren Featureanzahl erwartet werden. Ersteres hingegen ist durchaus überraschend.

Zur weiteren Analyse lohnt sich deshalb ein Blick auf die Gesamtanzahl der Knoten in den resultierenden Bäumen, gezeigt in Abbildung 6.34. Hier wird nun deutlich, warum der Füllgrad des kdb-trees angestiegen ist. Zwar ist nach wie vor die Anzahl der Knoten bei

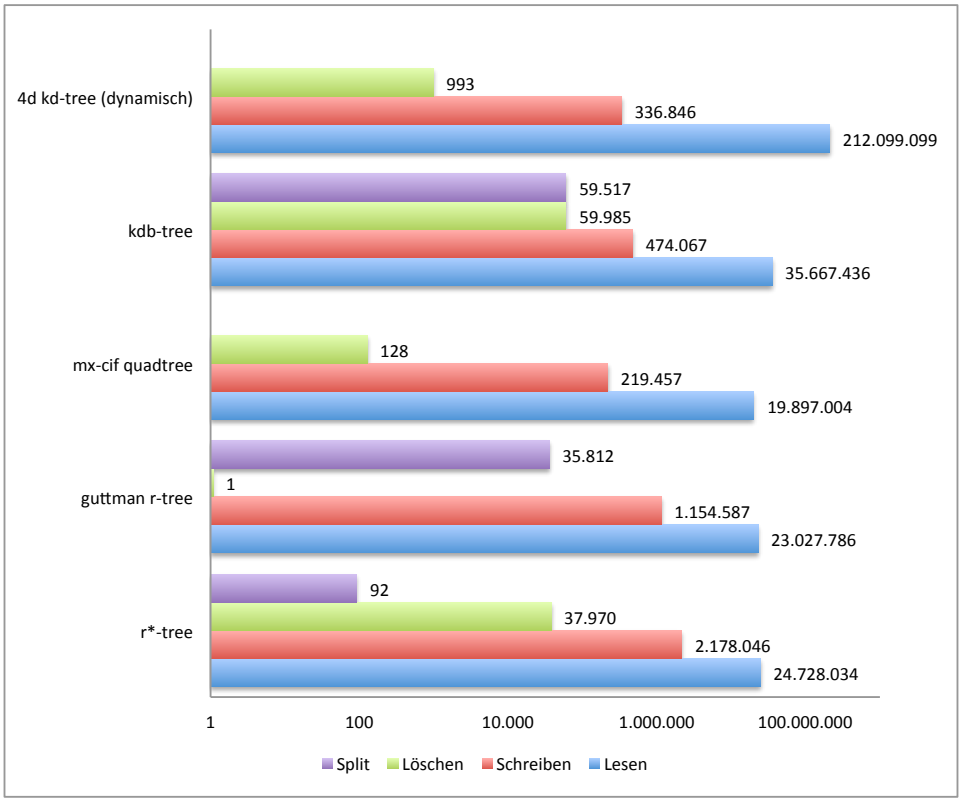

Abbildung 6.31: Anzahl der Operationen auf Knoten/Pages im ReadWrite-Test (Datensatz NRW - Gebäude)

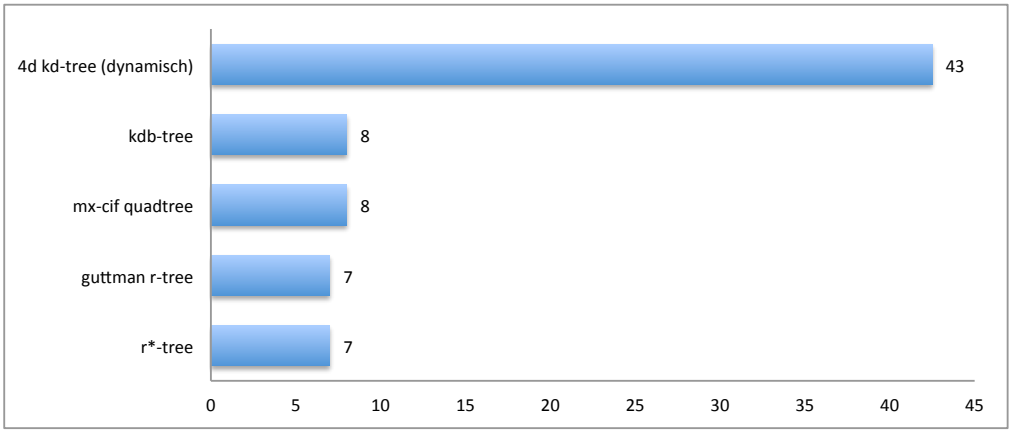

Abbildung 6.32: Höhe des Baums am Ende des ReadWrite-Tests (Datensatz NRW - Gebäude)

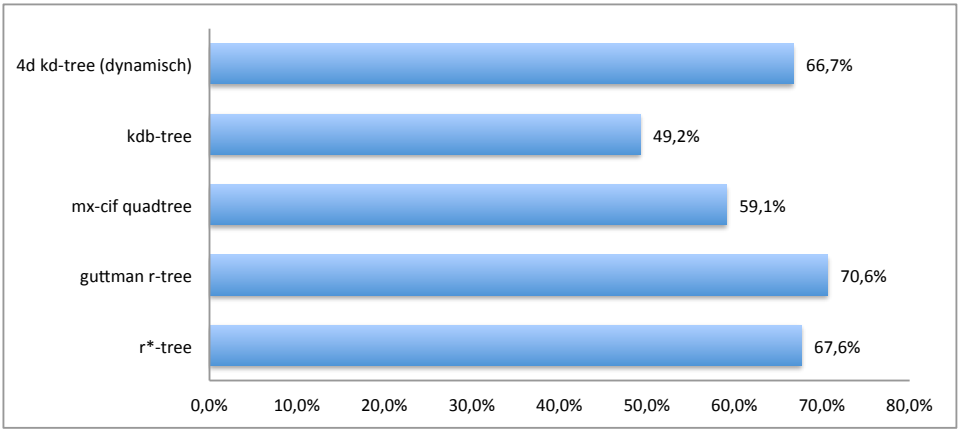

Abbildung 6.33: Füllgrad der Knoten am Ende des ReadWrite-Tests (Datensatz NRW - Gebäude)

diesem Index deutlich höher als bei den meisten anderen. Gemessen an der Gesamtanzahl der Features, welche in etwa der Anzahl der Knoten im four-dimensional kd-tree entspricht[5], ist die Zahl der Knoten im kdb-tree im Vergleich zum BuildIndex-Test geringer. Da die zufälligen Lese- und Schreiboperationen eher zu einer Verschlechterung der Qualität des Baums beitragen dürften ist die Ursache folglich eher in der insgesamt geringeren Anzahl der Features zu suchen.

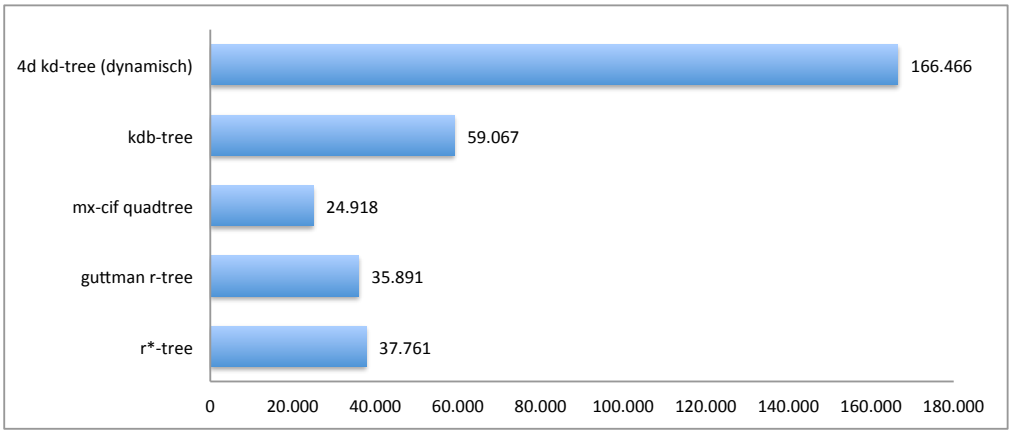

Abbildung 6.34: Anzahl der Knoten am Ende des ReadWrite-Tests (Datensatz NRW - Gebäude)

[5]Da beim ReadWrite-Test die Operationen nur mit gewissen Wahrscheinlichkeiten durchgeführt werden, unterscheidet sich die Anzahl der Features im Baum bei jedem Lauf. Die in der Abbildung gezeigten Werte sind die Mittelwerte als allen durchgeführten Läufen.

6.2.2.6 Fazit

Bei Verwendung von Nicht-Punktgeometrien zeigt sich noch deutlicher der Vorteil der r-trees. Bei Suchoperationen sind diese durchweg die schnellsten in den Tests gewesen. Insbesondere der str-tree konnte für statische Geodaten überzeugen. Letztlich waren nur zwei Schwachpunkte auszumachen. Erstens sind Einfüge- und Löschoperationen verhältnismäßig teuer. Zweitens sind beim guttman r-tree und r*-tree die Anzahl der I/O-Operationen sowie deren Umfang eher im Mittelfeld anzuordnen. Dies bedeutet, dass wenn sehr schnell ein Index erzeugt werden soll oder ein relativ langsames Speichermedium verwendet wird, ein anderer Index als ein r-tree durchaus seine Berechtigung hat. Welcher der getesteten aber alternativ genommen werden kann hängt dabei stark vom Einsatzzweck.

6.2.3 Bewegungsdaten

Zum Abschluss soll nun noch ein Blick auf das Verhalten der Indices bei Verwendung von Bewegungsdaten erfolgen. Hierzu wurde auf Basis des Datensatzes „Gummersbach - LineStrings“ ein entsprechender Straßen-Layer erzeugt. Auf die Verwendung des Datensatzes NRW wurde hingegen verzichtet, weil sich dieser als zu Umfangreich für das Simulationssystem erwiesen hat.

Da es sich bei den hier vorliegenden Bewegungsdaten um dynamische Punktgeometrien handelt, werden folglich die selben Indices berücksichtigt wie auch beim ReadWrite-Test für Punktgeometrien. Ein entsprechender Vergleich mit dem Verhalten bei gewöhnlichen dynamischen Daten ist folglich auch gegen diesen Test durchzuführen.

Insgesamt wurden zwei verschiedene Tests betrachtet, einer für Window Queries (BBoxes) und einer für Nearest Neighbor-Abfragen. In dieser Reihenfolge werden auch die entsprechenden Tests vorgestellt. Anschließend folgt noch ein kurzes Fazit.

6.2.3.1 Window Queries

Betrachtet man zu Anfang das Laufzeitverhalten (Abbildung 6.35), so fällt eine gewisse Ähnlichkeit zum ReadWrite-Test auf. Insbesondere ist das sehr gute Verhalten der r-trees bei Window-Queries zu erkennen. Auf der anderen Seite stehen wiederum die quadtrees, die hier eher schlechte Werte aufweisen. Das umgekehrte Bild zeigt sich wiederum auch bei den Schreiboperationen, nur das hier die quadtrees klar im Vorteil sind. Ebenso sind die Werte des kd-trees in beiden Bereichen sehr positiv. Überraschend ist dafür, dass auch der kdb-tree einen sehr guten Wert erzieht, bei Suchoperationen sogar einen signifikant besseren.

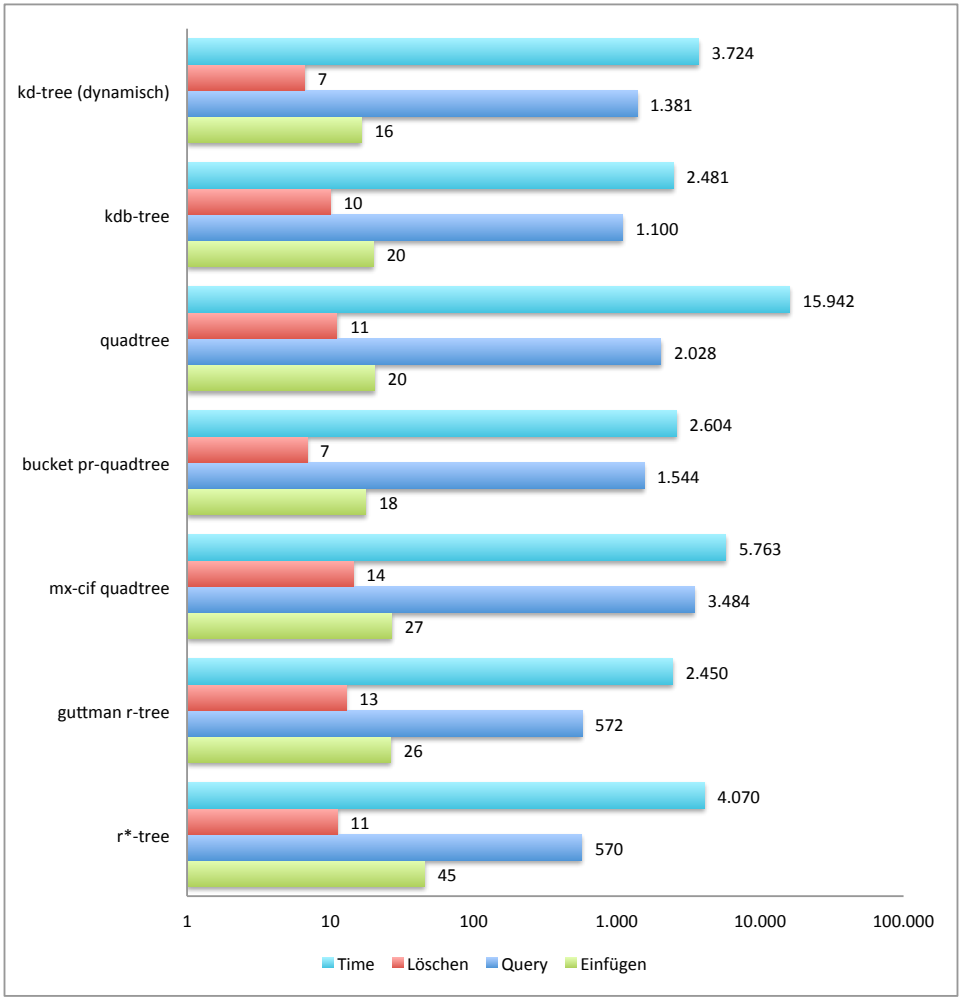

Abbildung 6.35: Laufzeiten der verschiedenen Operationen im BBox-Test (Datensatz Gummersbach - Straßen))

Zieht man nun die Anzahl der Page-Operationen hinzu (Abbildung 6.36), so fällt ein gewisser Unterschied auf. Sehr gute Werte können beim bucket pr-quadtree beobachtet werden, auch gerade bei den Split- und Löschoperationen. Dieser Index kommt folglich mit sehr wenigen umfangreicheren Modifikationen am Baum aus. Interessant ist zudem die sehr niedrige Anzahl an Leseoperationen des guttman r-trees, welche deutlich unter allen anderen liegt.

Wie auch schon zuvor ist bei diesen Zahlen jedoch die Pagegröße ebenfalls zu berücksichtigen, was in Abbildung 6.37 geschehen ist. Hierdurch relativieren sich vor allem die zu erst sehr negativ aussehenden Werte von kd-tree und quadtree. Allerdings fällt nun der mx-cif quadtree umso deutlicher bei den Leseoperationen auf. Dabei ist von dem selben Effekt auszugehen, der auch schon zuvor beschrieben wurde, nämlich der festen Höhe bei Punktgeometrien. Dieses Bild wird durch einen Blick auf die verschiedenen Höhen am

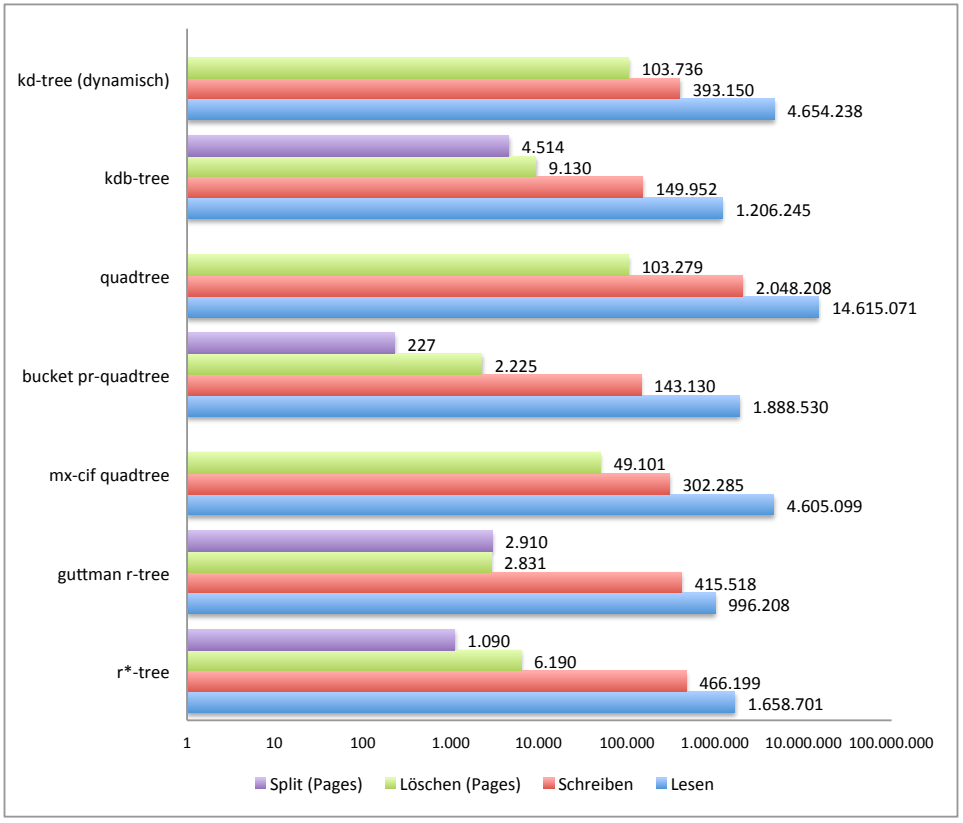

ABBILDUNG 6.36: Anzahl der Operationen auf Knoten/Pages im BBox-Test (Datensatz Gummersbach - Straßen)

Ende des Tests (Abbildung 6.38) auch bestätigt. Der mx-cif quadtree weißt hierbei mit einem Wert von acht eine doppelt so große Höhe auf wie die die balancierten Indices.

Auch bei der Pageanzahl setzt sich dieses Bild wiederum fort (Abbildung 6.36). Die Zahlen sind vor allem problematisch, wenn man sich vor Augen hält, dass die Anzahl der Knoten im kd-tree sowie quadtree in etwa der Anzahl an Features im mx-cif quadtree entspricht. Folglich besitzt dieser fast doppelt so viele Knoten wie tatsächlich Features.

6.2.3.2 Nearest Neighbor Queries

Für die Analyse der Messergebnisse des Nearest-Neighbor Tests genügt nun eine Beschränkung auf die Zahlen der eigentlichen Suchoperation, da sich die Schreiboperationen naturgemäß zum vorherigen Test gleichen. Dies ist auch in den Ergebnissen gut zu sehen, wobei wiederum die Messung der Laufzeiten den Anfang macht (Abbildung 6.40).

Deutlich wird dabei sofort wiederum die gewisse Ähnlichkeit zu den Ergebnissen des ReadWrite-Tests. Dieses Mal zeigen sich insbesondere die für die Suche des nächsten Nachbarn guten Eigenschaften des bucket pr-quadtrees, welcher einen leicht besseren

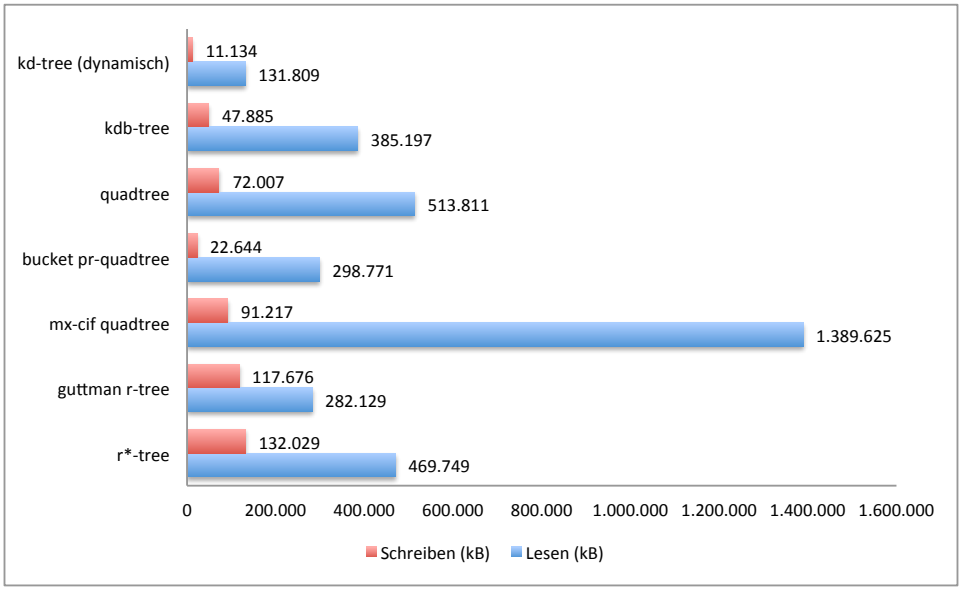

ABBILDUNG 6.37: Anzahl der gelesenen/geschriebenen Kilobytes im BBox-Test (Datensatz Gummersbach - Straßen)

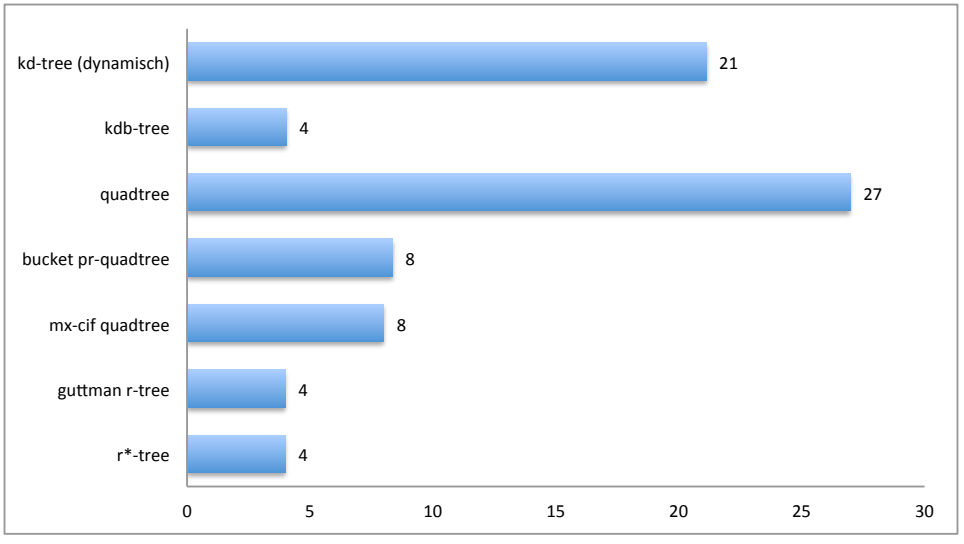

ABBILDUNG 6.38: Höhe des Baums am Ende des BBox-Tests (Datensatz Gummersbach - Straßen)

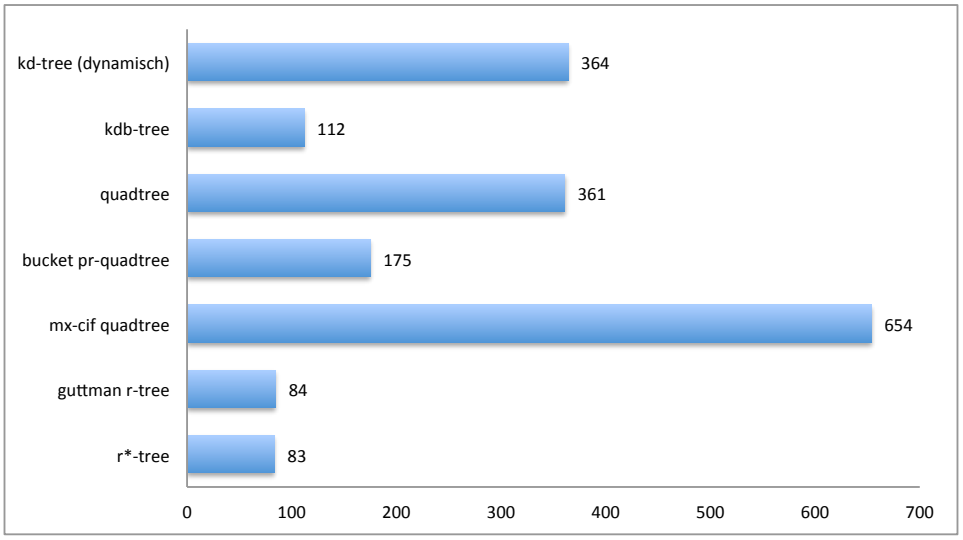

ABBILDUNG 6.39: Anzahl der Knoten am Ende des BBox-Tests (Datensatz NRW - Gebäude)

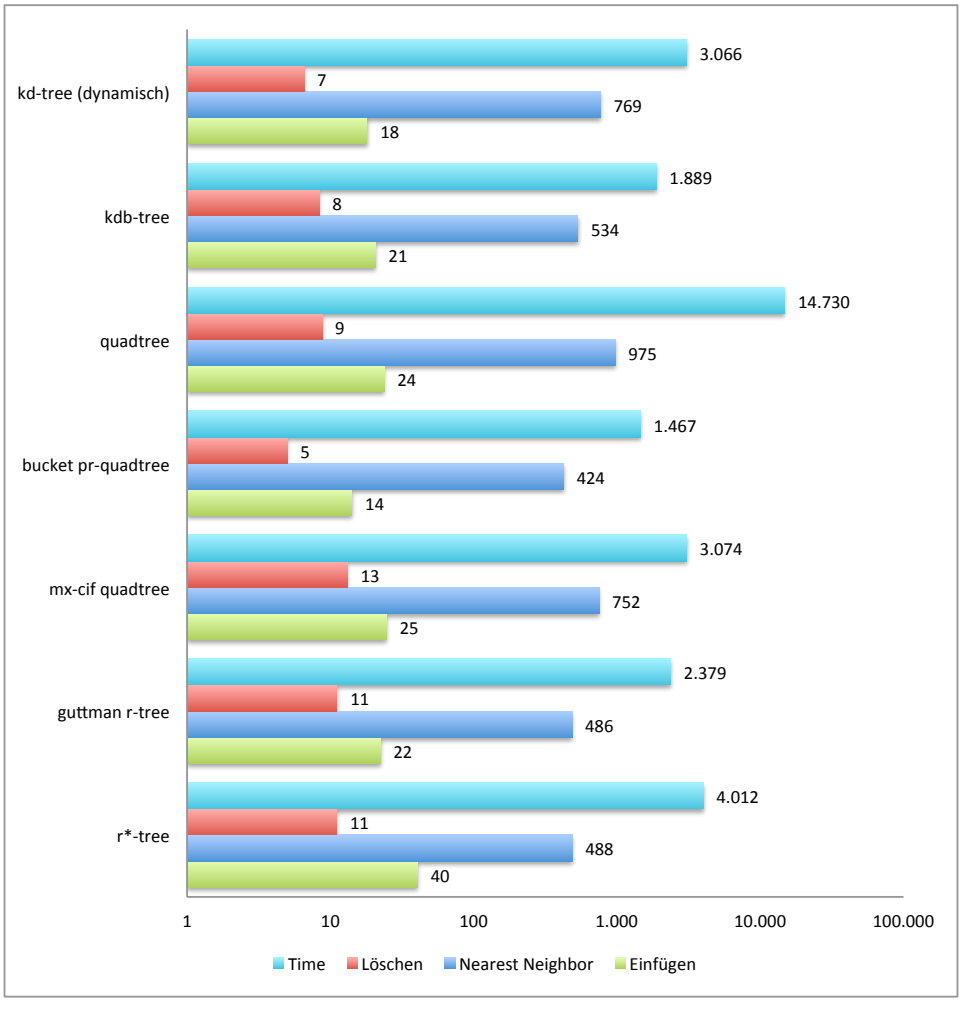

ABBILDUNG 6.40: Laufzeiten der verschiedenen Operationen im Nearest Neighbor-Test (Datensatz Gummersbach - Straßen))

Wert erzielt als die r-trees. Abweichend gestaltet sich allerdings der Rest der Zahlen. Besonders auffällig ist wiederum das gute Ergebnis des kdb-trees. Auf der anderen Seite weißt nun der quadtree den schlechtesten Wert auf.

Als ergänzend stellen sich auch hier wiederum die Anzahl der Knotenoperationen dar (Abbildung 6.36). Sehr gut zu erkennen sind dabei die positiven Werte von kdb-tree, guttman r-tree und bucket pr-quadtree, welche auch unter Berücksichtigung der Pagegröße als gut zu bewerten sind. Auf der anderen Seite steht der klassische quadtree mit einer sehr hohen Anzahl an Leseoperationen.

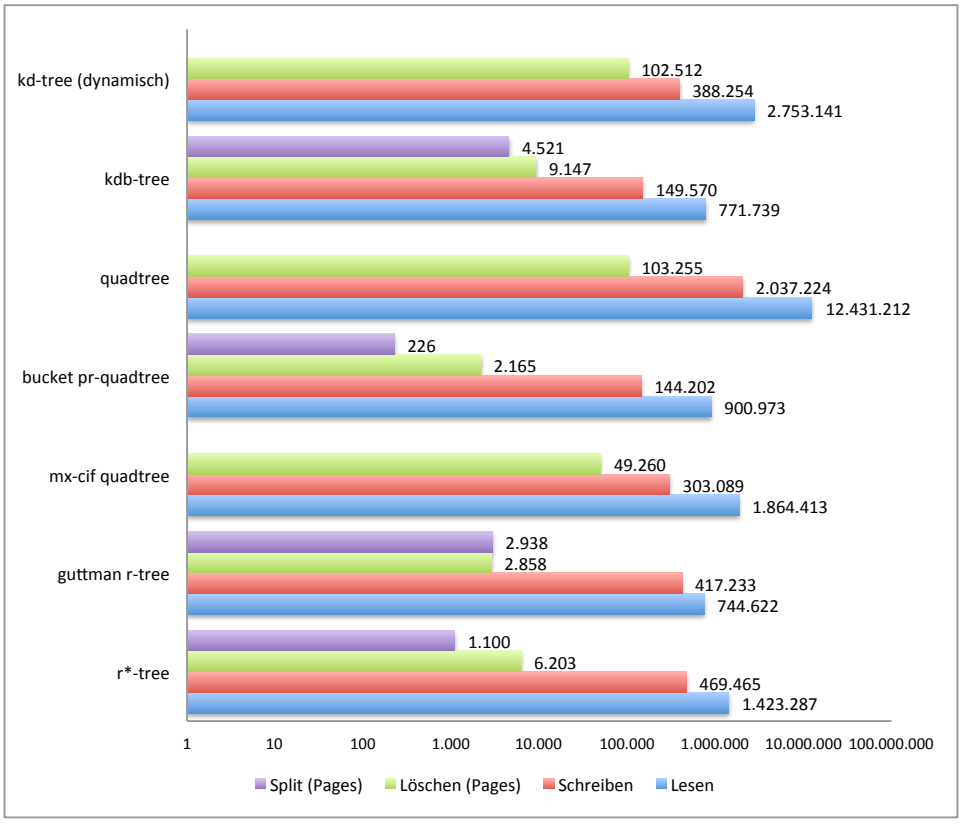

Abbildung 6.41: Anzahl der Operationen auf Knoten/Pages im Nearest Neighbor-Test (Datensatz Gummersbach - Straßen)

6.2.3.3 Fazit

Auch bei den Bewegungsdaten setzt sich der Trend der positiven Ergebnisse für die r-trees fort. Ebenso konnte für Nearest Neighbor-Operationen mit einen guten Wert des bucket-pr quadtrees gerechnet werden. Positiv ist hingegen in beiden Tests der bisher eher negativ aufgefallene kdb-tree in Erscheinung getreten, was in dieser Form sicherlich nicht zu erwarten war.

Insgesamt zeigen sich nur geringfügige Abweichungen zwischen „normalen“ dynamischen Daten und Bewegungsdaten. Daher dürfte es in vielen Fällen ausreichen einen entsprechend geeigneten Index für reguläre dynamische Daten zu wählen. Nur in speziellen Situationen lohnt sich ein weitergehender Blick auf die Verhaltensweisen bei dieser Form der Geodaten.

Kapitel 7

Fazit

Das zu Anfang gestellte Ziel eines Vergleichs verschiedener Indextypen für unterschiedliche Arten von Geodaten konnte prinzipiell erreicht werden. Es wurde gezeigt, welche Konzepte bei welchen Operationen von Vorteil sind und wo entsprechende Nachteile zu erkennen sind. Schlussendlich wurde auch festgestellt, dass die große Verbreitung gerade der r-trees ihre Berechtigung zu haben scheint. Gleichwohl konnte der hier angestellte Vergleich bedingt durch die enorme Vielschichtigkeit nur einen kleinen Einblick gewähren. Er kann daher nur als Indiz verstanden werden bei der Auswahl geeigneter Indices.

Nicht verschwiegen werden sollen auch die Probleme, die auf dem Weg dorthin aufgetreten sind. Hier ist insbesondere der kdb-tree zu nennen, der bereits in Kapitel 6 des häufigeren genannt wurde. Hier muss die Entscheidung diesen Index für Nicht-Punktgeometrien zu erweitern rückblickend klar negativ beurteilt werden. Wenn auch das Konzept interessant ist, so hat sich in der Umsetzung gezeigt, dass es eine fast unüberschaubare Anzahl an Problemen und Fehlern produziert, die zum Teil nur unter sehr bestimmten (seltenen) Situationen aufgetreten sind. Damit einher ging eine nicht zu unterschätzende Verzögerung der gesamten Thesis. Zugleich sind auch die Ergebnisse dieses Index nur eingeschränkt anwendbar, da durch die Modifikationen durchaus gewichtige Unterschiede zur ursprünglichen nur für Punktgeometrien entwickelten Variante aufgetreten sind. Dabei ist vor allem die gestiegene Pagegröße zu nennen.

Auf der anderen Seite sollen aber auch die beiden für den Autor positivsten Entdeckungen kurz erwähnt werden. Wenn gleich nicht sonderlich schnell, so besitzt der four-dimensional kd-tree in seinem Lösungsansatz doch einen gewissen Reiz, da dieser die Erweiterung eines eigentlich rein auf Punktgeometrien ausgelegten Verfahrens auf entsprechende Flächengeometrien mit denkbar wenigen Änderungen ermöglicht. Hier wäre es sicherlich von

Interesse gewesen eine entsprechende Änderung des ursprünglichen kdb-trees zu untersuchen. Als zweite Entdeckung ist der bucket pr-quadtree zu nennen, welcher gerade bei Nearest Neighbor-Abfragen ein sehr gutes Laufzeitverhalten gezeigt hat und zudem durch seine sehr einfache Implementierung zu überzeugen weiß.

Mit einem gewissen Fragezeichen ist der r*-tree zu beurteilen. Die hier ermittelten Messwerte sind zwar prinzipiell nicht schlecht und ähnlich denen des guttman r-trees. Die Erwartungshaltung war jedoch ein messbar besseres Laufzeitverhalten, welches nicht angetroffen werden konnte. Es besteht zumindest die Möglichkeit, dass hier ein noch nicht entdeckter Fehler in der Implementierung die Ursache ist. Gleichwohl kommen auch die gewählte Bucketgröße wie auch das verwendete Datenmaterial in Frage. Möglicherweise war aber auch die Erwartungshaltung falsch. In jedem Fall wären hier weitere Analysen angebracht bevor ein entsprechender Schluss gezogen werden kann.

Neben der Überprüfung der Messwerte ließe sich, wie bereits beim kdb-tree erwähnt, mit entsprechender Zeit diese Thesis inhaltlich sinnvoll erweitern und komplettieren. Denkbar wären dabei die folgenden Bereiche:

- Die Analyse des Verhaltens beim Einsatz von Caching, welches gerade für die Praxis ein sehr wichtiger Aspekt sein dürfte.
- Die Optimierung der verwendeten Parameter, beispielsweise die Knotengrößen oder beim mx-cif quadtree die maximale Baumhöhe. Besonders bei letzterem konnte bereits deutlich gezeigt werden, wie sich diese auf das Laufzeitverhalten auswirken kann.
- Die Betrachtung komplexerer Operationen wie Joins, welche naturgemäß deutlich andere umfangreichere Anforderungen an die Indices stellen, allerdings auch in der Praxis entsprechend seltener anzutreffen sind.
- Die Erweiterung um zusätzliche Indices. Es ist dabei jedoch zu bedenken, dass die Betrachtung aller möglichen Indices bedingt durch die unüberschaubare Menge an Varianten im vollen Umfang nicht durchführbar ist.

Daneben bietet sich noch ein weiterer Bereich an, der mehr ein vollständig neuer Anwendungsbereich denn eine Erweiterung des hier behandelten ist. Gemeint ist damit die Betrachtung temporaler Geodaten, beispielsweise Fahrzeugbewegungen mit Indizierung der gesamten Route und nicht nur der momentanen Position (wie hier geschehen). Damit einher gehen auch andere Indices, um die temporale Veränderung der Eigenschaften entsprechend sinnvoll berücksichtigen zu können. Gleichwohl wäre es über das verknüpfende Element der Bewegungsdaten ein sehr interessantes Themenfeld.

Als letztes bleibt noch bilanzieren, dass die Thesis sehr interessant und aufschlussreich war. Ich hoffe, dass dies auch für den Leser zutrifft und bedanke mich für die entgegengebrachte Aufmerksamkeit.

Literatur

Beckmann, *Norbert* u. a.: The R*-tree: an efficient and robust access method for points and rectangles, in: Proceedings of the 1990 ACM SIGMOD international conference on Management of data (SIGMOD '90), Atlantic City, New Jersey, United States 1990, S. 322–331.

Brinkhoff, *Thomas*: A Framework for Generating Network-Based Moving Objects, in: Geoinformatica 6 (2 2002), S. 153–180.

Brinkhoff, *Thomas*: Geodatenbanksysteme in Theorie und Praxis, Heidelberg 2008.

Ciaccia, *Paolo*, *Marco Patella* und *Pavel Zezula*: M-tree: An Efficient Access Method for Similarity Search in Metric Spaces, in: Proceedings of the 23rd International Conference on Very Large Data Bases (VLDB '97), San Francisco, CA, USA 1997, S. 426–435.

Environmental Systems Research Institute, Inc: ESRI Shapefile Technical Description, Juli 1998.

Guttman, *Antonin*: R-trees: a dynamic index structure for spatial searching, in: Proceedings of the 1984 ACM SIGMOD international conference on Management of data (SIGMOD '84), Boston, Massachusetts 1984, S. 47–57.

ISO 19107:2003 Geographic information – Spatial schema, Techn. Ber., International Organization for Standardization (TC 211), 2003.

Kedem, *Gershon*: The quad-CIF tree: A data structure for hierarchical on-line algorithms, in: Proceedings of the 19th Design Automation Conference (DAC '82), Piscataway, NJ, USA 1982, S. 352–357.

Leutenegger, *Scott T.*, *Jeffrey M. Edgington* und *Mario A. Lopez*: STR: A simple and efficient algorithm for R-tree packing, Techn. Ber., 1997.

Manolopoulos, *Yannis* u. a.: R-Trees: Theory and Applications (Advanced Information and Knowledge Processing), 1. Aufl., Sep. 2005.

Navarro, *Gonzalo*: Searching in metric spaces by spatial approximation, in: The VLDB Journal 11 (1 2002), S. 28–46.

Robinson, *John T.*: The K-D-B-tree: a search structure for large multidimensional dynamic indexes, in: Proceedings of the 1981 ACM SIGMOD international conference on Management of data (SIGMOD '81), Ann Arbor, Michigan 1981, S. 10–18.

Samet, *Hanan*: Foundations of Multidimensional and Metric Data Structures (The Morgan Kaufmann Series in Computer Graphics and Geometric Modeling), San Francisco, CA, USA 2005.

Erklärung

Ich versichere, die von mir vorgelegte Arbeit selbständig verfasst zu haben. Alle Stellen, die wörtlich oder sinngemäß aus veröffentlichten oder nicht veröffentlichten Arbeiten anderer entnommen sind, habe ich als entnommen kenntlich gemacht. Sämtliche Quellen und Hilfsmittel, die ich für die Arbeit benutzt habe, sind angegeben. Die Arbeit hat mit gleichem Inhalt bzw. in wesentlichen Teilen noch keiner anderen Prüfungsbehörde vorgelegen.

Ort: _____

Datum: _____

Unterschrift: _____